VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Generování asociačních pravidel pro velká data

Association Rules Generation for Big Data

2017 Bc. Vojtěch Kotík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Vojtěch Kotík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Generování asociačních pravidel pro velká data
Association Rules Generation for Big Data

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem projektu je implementovat jednu nebo více metod pro generování asociačních pravidel a upravit je pro efektivní běh v paralelním prostředí nad velkými daty. Paralelizace bude řešena pomocí OpenMP, MPI nebo CUDA. Výsledné algoritmy budou otestovány nad dodanými daty a výsledky analýzy budou přehledně zobrazeny a vyhodnoceny. Implementace bude provedena v jazyce C++.

Práce bude obsahovat:
1. Seznámení s problematikou.
2. Aktuální State of the art.
3. Podrobný popis zvoleného/zvolených algoritmů.
4. Experimenty, měření, vyhodnocení.
5. Závěr - zhodnocení výsledků.

Seznam doporučené odborné literatury:

[1] Mohammed J. Zaki, Wagner Meira, Jr., Data Mining and Analysis: Fundamental Concepts and Algorithms, Cambridge University Press, May 2014. ISBN: 9780521766333.
[2] Aggarwal C.C. (2015), Data Mining: The Textbook, Springer

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26.dubna 2017 .....

Rád bych poděkoval vedoucímu diplomové práce doc. Ing. Janu Platošovi, Ph.D. za odbornou pomoc a přínosné konzultace objasňující nejasnosti vzniklé při vytváření této práce. Také bych rád poděkoval Ing. Haně Kaniové a Mgr. Matěji Veselému za podporu a motivaci.

Abstrakt

Asociační pravidla jsou nástrojem analýzy dat, který umožňuje nacházet vztahy mezi objekty. Jedním z typických využití je analýza nákupních košíků v obchodě pro lepší optimalizaci prodejních taktik. Objem analyzovaných dat stále roste. Dříve navržené algoritmy pro generování asociačních pravidel již nereflektují možnosti současných systémů a proto je nutné je upravit. Tato diplomová práce se zabývá popsáním vybraných algoritmů a jejich úpravou pro efektivní běh v paralelním prostředí velkých dat.

Klíčová slova: asociační pravidla, apriori, eclat, FP Growth, velká data, paralelismus

Abstract

Association rules are data analysis tool which can find relationships between objects. Shopping cart analysis for supermarket optimalization is one of the typical use cases. Volume of data for analysis is constantly growing. Older algorithms for association rules generation cannot fully utilize computational capabilities of state of the art systems and. Adjustment is required. This diploma thesis describes those algorithms and their modification for big data parallel computing.

Key Words: association rules, apriori, eclat, FP Growth, big data, parallelism

Obsah

Seznam použitých zkratek a symbolů

sup	–	support
conf	–	confidence
FPG	–	algoritmus FP Growth

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Asociační pravidla jsou velmi účinným nástrojem pro reprezentaci vzorů chování nalezených v datech. Jedním z původních možných využití je analýza nákupních transakcí zákazníků v obchodech pro lepší predikci jejich nákupního chování, která umožní obchodníkům optimalizovat jejich prodejny a v konečném důsledku zvýšit tržby.

Základní algoritmy pro generování těchto asociačních pravidel byly vytvořeny již v minulém století. Nedokáží tedy využít nové výpočetní schopnosti, jako jsou například paralelní výpočty. Tato diplomová práce se zabývá popisem a optimalizací těchto algoritmů pro běh v paralelním prostředí, což umožní efektivně zpracovávat velmi velké datasety, známé jako big data.

V úvodní kapitole 2 popíšeme problematiku a základní algoritmy. Poté v kapitole 3 popíšeme aktuální State of the art vylepšení těchto algoritmů. V kapitole 4 následuje detailní popis algoritmů vybraných pro implementaci. Samotná implementace je popsána v kapitole 5 a následné experimenty nad testovacími daty v kapitole 6. V závěrečné kapitole 7 výsledky těchto experimentů vyhodnotíme.

Tato práce rozvíjí můj semestrální projekt věnovaný totožnému tématu z předcházejícího školního roku.

Součástí je funkční aplikace pro generování asociačních pravidel včetně zdrojových kódů, které lze nalézt na přiloženém DVD.

2 Seznámení s problematikou

Problém generování asociačních pravidel původně vzniknul v prostředí supermarketů. Obvyklý supermarket obsahuje velký počet různých druhů zboží. Marketingová oddělení těchto obchodů často stojí před úkolem, jak optimalizovat prodej tak, aby zákazník utratil v jejich obchodě co nejvíce peněz. Dosáhnout toho lze různými způsoby, jako je změna rozmístění zboží nebo vhodně nastavené slevové akce. Pro příznivý výsledek těchto akcí je nutné poznat a zmapovat co nejpřesněji nákupní zvyky zákazníků. K tomuto účelu začaly obchody sbírat a uchovávat data o transakcích zákazníků. Informace popisované v této kapitole byly čerpány především z [1] a [2].

Mezi možná využití asociačních pravidel patří:

- **analýza nákupního košíku** - původní problém stojící za vznikem této oblasti. To je také důvodem, proč terminologie vychází právě z prostředí supermarketů, jako je např. **itemset** - kolekce položek (item) zakoupených zákazníkem.
- **analýza textu** - vzhledem k častému opakování slov a frází v textech lze asociační pravidla použít k identifikaci výrazů, které se často vyskytují společně a také u klíčových slov
- **predikce kriminality** - na základě údajů o městech, demografii a historii zločinů lze predikovat riziko opakování těchto zločinů a lépe porozumět jejich příčinám
- **pojišťovnictví** - pojišťovny mohou stanovit přesnější skóringové modely pro určitá rizika na základě analýzy předchozích pojistných událostí
- **bankovní sektor** - na základě analýzy údajů o splácení úvěrů je možné zpřesnit odhady rizika u nových žadatelů
- **počítačová bezpečnost** - jednou z možností je odhalení možného napadení infrastruktury pomocí analýzy bezpečnostních logů aplikací

Asociační pravidla se značí formou implikací: X $\Rightarrow$ Y. X a Y jsou itemsety. Klasickým příkladem je pravidlo Pivo $\Rightarrow$ Dětské plenky. Toto pravidlo říká, že když si zákazník koupí pivo, tak s velkou pravděpodobností zakoupí také dětské plenky. Tento vzor chování byl jeden z prvních objevených touto metodou a je možné jej využít pro tvorbu personalizovaných slevových kuponů cílených na čerstvé rodiče.

2.1 Reprezentace dat získaných z nákupů

2.1.1 Binární dataset

Základní reprezentace těchto dat je obvykle ve formě binárního datasetu **D**, který tvoří matici vztahu mezi transakcemi a položkami. Položky I tvoří sloupce matice. Každý sloupec odpovídá unikátní položce nabízené v obchodě. Transakce T je složena z identifikátoru a položek I, které

Tabulka 1: Binární dataset [1]

D	A	B	C	D	E
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

Tabulka 2: Transakční dataset [1]

t	i(t)
1	ABDE
2	BCE
3	ABDE
4	ABCE
5	ABCDE
6	BCD

daný nákup obsahoval. Každý řádek reprezentuje jeden nákup (transakci). Pokud daný nákup obsahoval danou položku, je v průsečíku TxI jednička. Ukázka binárního datasetu je v tabulce 1. I = A,B,C,D,E a T = 1,2,3,4,5,6.

Délka každého řádku binárního datasetu odpovídá celkovému počtu sledovaných položek a obsahuje spoustu nulových záznamů. Proto v praxi obsahuje téměř samé nuly a tento způsob uložení dat má velkou paměťovou náročnost. Proto byly vyvinuty úspornější modely.

2.1.2 Transakční dataset

Transakční dataset je vylepšením binárního. Uložení zůstává horizontální, ale v jednotlivých řádcích jsou nyní identifikátory položek. Položky, které daný nákup neobsahuje již nyní nezabírají cenný paměťový prostor. Ukázka transakčního datasetu odpovídajícímu binárnímu z předchozího příkladu je v tabulce 2. První transakce nyní bude <1, {A,B,D,E}>. Položka C v této transakci není obsažena, proto ji vynecháme.

2.1.3 Vertikální dataset

Transakční model byl ještě vylepšen do podoby vertikálního datasetu. V něm počet sloupců odpovídá počtu položek, které může nákup obsahovat. Sloupec obsahuje pouze identifikátory nákupů, ve kterých byla tato položka obsažena. Nejdelší možná délka sloupce odpovídá počtu analyzovaných transakcí, pokud všechny obsahovaly tuto položku. Ukázka vertikálního datasetu je v tabulce 3. Položka A byla zakoupena pouze v nákupech 1,3,4 a 5, proto má příslušný řádek formu <A,{1345}>.

Tabulka 3: Vertikální dataset [1]

x	A	B	C	D	E
t(x)	1 3 4 5	1 2 3 4 5 6	2 4 5 6	1 3 5 6	1 2 3 4 5

2.2 Frequent pattern mining a support

Nejrozšířenější model pro generování asociačních pravidel používá tzv. **support**(četnost).

Cílem je identifikace všech skupin položek, které se vyskytují v nákupech společně. Při požadavku na generování je uživatelem definována minimální hodnota supportu, které musí vygenerované množiny položek splňovat. Například můžeme definovat, že nás zajímají jen skupiny s minimálním supportem 0.3 (30% všech nákupů). Poté generujeme všechny možné množiny rozměru k. K nabývá velikost od 1 do počtu všech unikátních položek. Množiny splňující podmínku minimálního supportu se nazývají **frequent itemsety** *F*. Z datasetu v tabulce 1 při supportu 0.3 získáme tyto frequent itemsety:

$F^(1) = \{A,B,C,D,E\}$

$F^(2) = \{AB,AD,AE,BC,BD,BE,CE,DE\}$

$F^(3) = \{ABD,ABE,ADE,BCE,BDE\}$

$F^(4) = \{ABDE\}$

2.3 Association rules mining a confidence

Nechť A a B jsou skupiny zboží. Pravidlo A ⇒ B říká, že pokud zákazník zakoupí všechno zboží ze skupiny A, pak zakoupí také všechno zboží ze skupiny B. Toto pravidlo bude považováno za platné, pokud A i B splní podmínky minimálního supportu a toto pravidlo splňuje uživatelem definovanou podmínku minimální **confidence**.

Confidence je definováno jako podmíněná pravděpodobnost:

$$c = \text{conf}(A \Rightarrow B) = P(A|B) = \frac{P(A \wedge B)}{P(B)} = \frac{support(AB)}{support(A)}$$

Při použití pravidla BC ⇒ E bude výpočet probíhat takto:

$$s = \text{sup}(BC \Rightarrow E) = \text{sup}(BCE) = 3$$

$$c = \text{conf}(BC \Rightarrow E) = \frac{sup(BCE)}{sup(BC)} = 3/4 = 0.75$$

2.4 Algoritmy pro generování asociačních pravidel

Asociační pravidla musí splňovat podmínky minimálního supportu a minimální confidence. Z toho důvodu algoritmy tyto pravidla generující obvykle obsahují dvě fáze:

1. Ověření splnění minimálního supportu a vygenerování všech tuto podmínku splňujících **frequent itemsetů**
2. Vygenerování všech pravidel z frequent itemsetů a ověření podmínky splnění minimální confidence

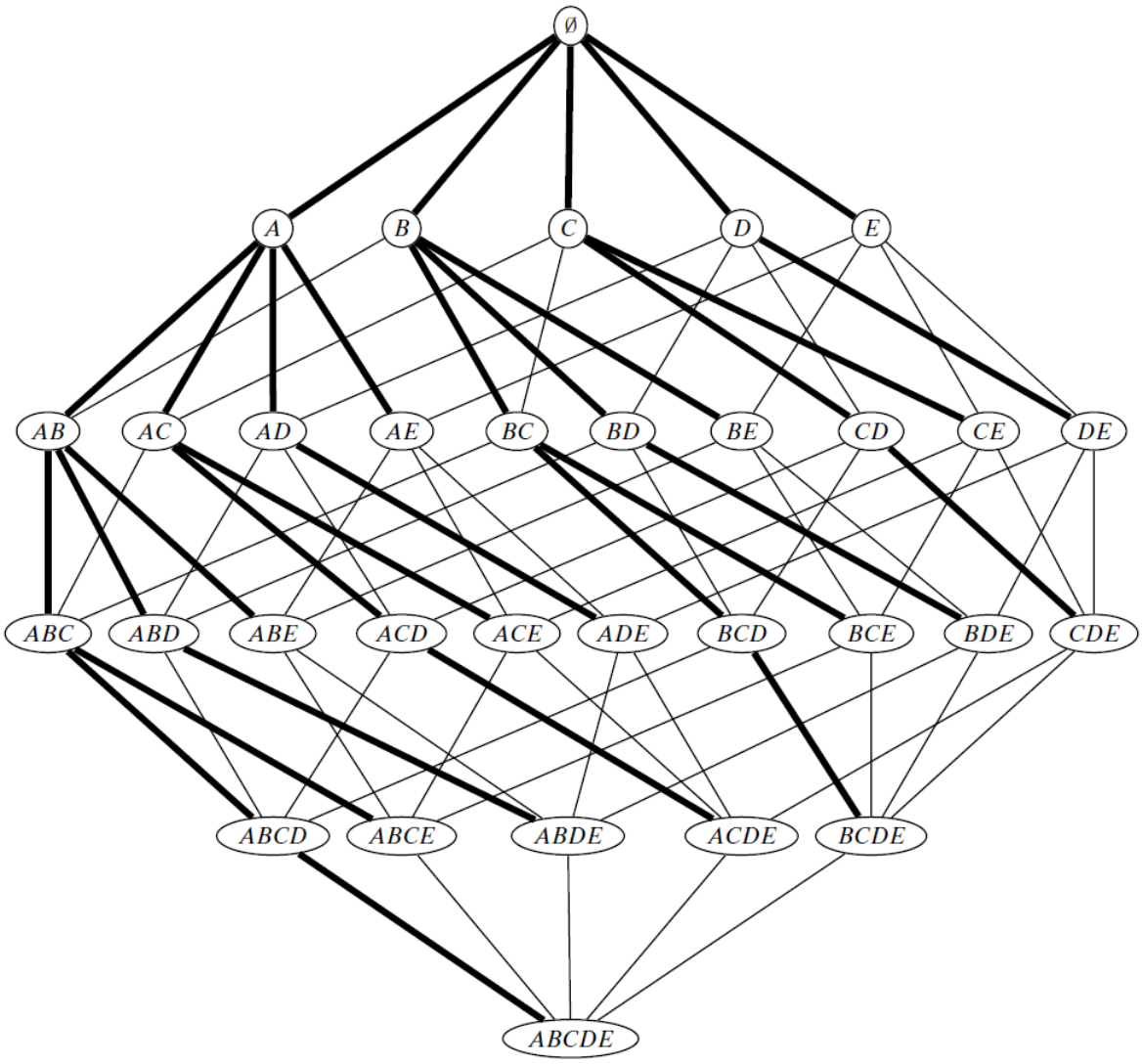
První fáze ověření supportu trvá mnohem déle než druhá a proto byla většina snahy o vylepšení algoritmů pro generování směřována právě do této fáze. Vybrané algoritmy popsané v této kapitole jsou popsány detailně v kapitole 4.

2.4.1 Brute Force

Pro kolekci položek obchodu I existuje 2^I - 1 možných kombinací, které mohou být potenciálně frequent itemsety, tzv. **kandidáti**. Algoritmus brute force vygeneruje všechny možné kombinace všech položek. Jejich vygenerováním vzniká základní možnost ověření, která pokračuje výpočtem supportu oproti každé transakci datasetu. Tento způsob je však velmi výpočetně náročný. Již v případě stovky možných položek získáme 2^{100} kandidátů a ověření takového počtu je problematické i při malém počtu transakcí. Ukázka algoritmu pro dataset z tabulky 1 je na obr. 1.

Lepších výsledků je možno dosáhnout jednou z následujících úprav:

- zmenšení počtu kandidátů, pro které je nutno vypočítat support
- efektivnější výpočet supportu zahrnutím pouze těch transakcí, které jsou pro dané položky relevatní
- použití kompaktnějších datových struktur



Obrázek 1: Algoritmus Brute Force [1]

2.4.2 Apriori

Cílem algoritmu apriori je zmenšení počtu generovaných kandidátů, pro něž je nutné vypočítat support. Je založen na myšlence, že *každá podmnožina frequent itemsetu automaticky také splňuje podmínku minimálního supportu*. Toho je využito k zefektivnění generace kandidátů. Když kandidát rozměru k nesplňuje podmínku minimálního supportu, tak nemá smysl z něj dále generovat další kandidáty o velikosti $k+1$ a větší.

Apriori začíná od nejmenších kandidátů velikosti 1. Po výpočtu supportu dojde k vytřídění kandidátů splňujících minimální support a jen z nich jsou následně generováni kandidáti úrovně 2. Takto pokračuje postupné rozšiřování velikosti kandidátů po jednotlivých úrovních, dokud již není možné vygenerovat žádné nové.

Tento algoritmus je detailně popsán v kapitole 4.1

2.4.3 Eclat

Algoritmus Eclat patří do skupiny algoritmů, které používají vertikální reprezentaci datasetu. Každý sloupec reprezentující danou položku zboží obsahuje identifikátory nákupních transakcí, ve kterých byla zakoupena. Proto ulehčuje náročné počítání supportu, které zde probíhá ve formě výpočtu průniku sloupců, obsahujících položky aktuálně zpracovávaného itemsetu. Průnikem identifikátorů dvou položek získáme množinu identifikátorů transakcí, které obsahují obě položky. Délka této množiny je hodnotou support pro tento průnik.

Algoritmus Eclat je založen na starších algoritmech Partition a Monet a detailně popsán v kapitole 4.2.

2.4.4 Rekurzivní algoritmy

Používají kompaktní datové struktury **enumeration tree**, nejčastěji v podobě **FP stromu**. Tento přístup je založen na myšlence efektivnějšího výpočtu supportu zahrnováním pouze relevantních transakcí. Toho je dosaženo převodem datasetu do stromové struktury.

Možnosti implementace:

- pole bez ukazatelů - použijeme pole pro reprezentaci datasetu a jeho podmnožin
- pole s ukazateli, ale bez FP stromu - umožní rozdělit dataset do menších polí a ty navzájem propojit pomocí ukazatelů.
- FP strom s ukazateli - detailně popsáno v kapitole 4.3

2.5 Alternativní koeficienty četnosti

Klasický výpočet důležitosti pravidla pomocí supportu a confidence má své limity. Může totiž vygenerovat pravidla, která sice splní definované hodnoty, ale ze statistického hlediska jsou

nevýznamné. Proto byly navrhnuty algoritmy založené na mírách podobnosti z oblasti statistiky. Některé z nich jsou definovány pouze pro množiny rozměru 2.

2.5.1 Korelační koeficient

Pearsonův korelační koeficient je pro vztah mezi proměnnými X a Y definován:

$$\rho = \frac{E[X \cdot Y] - E[X] \cdot E[Y]}{\sigma(X) \cdot \sigma(Y)}$$

U binárního datasetu jsou těmito hodnotami X a Y binární proměnné, které znázorňují (ne)přítomnost položky v transakci. E[X] je střední odhad proměnné X a σ(X) je směrodatná odchylka X. Relativní support položek definujeme jako *sup(i)* a *sup(j)* a relativní support jejich průniku jako *sup({i,j})*. Korelační koeficient pak vypočteme:

$$\rho_{ij} = \frac{sup(\{i,j\}) - sup(i) \cdot sup(j)}{\sqrt{sup(i) \cdot sup(j) \cdot (1 - sup(i)) \cdot (1 - sup(j))}}$$

Tento korelační koeficient může nabývat hodnoty z intervalu [-1,1]. Hodnota 1 znamená pozitivní korelaci, kdežto -1 negativní. Hodnoty okolo nuly znamenají slabou korelaci.

Pearsonův korelační koeficient je nejrobustnější statistická metoda pro posouzení závislosti mezi položkami. Není však jednoduché jeho výsledky správně posoudit u položek s nízkými hodnotami supportu.

Mezi další metody možného statistického posouzení závislosti patří např. χ^2, Cosinova podobnost nebo Jaccardův koeficient.

2.5.2 Big data

Jedna z definic pro big data říká, že jde o soubory dat velikosti takové, která je mimo schopnosti zpracování běžným softwarem v rozumném čase. V kontextu generování asociačních pravidel jde o datasety, jejich velikost výrazně přesahuje možnosti dříve navržených algoritmů.

Tím vznikly dva problémy:

- Všechny výše popsané algoritmy vznikly v době, kdy vývoj hardwaru směřoval cestou zvyšování frekvence jednojádrových procesorů. Proto jsou navrženy pro sekvenční běh. Později se však vývoj vydal směrem zvyšování počtu výpočetních jader procesorů, což umožnilo efektivní paralelní běh výpočtů pomocí více vláken.
- Datasety svou velikostí mohou převyšovat kapacitu dostupné operační paměti výpočetního stroje.

Úpravu některých klasických algoritmů pro běh v paralelním prostředí popisuje kapitola 5.

2.5.3 Pomocné algoritmy

Pro řešení problému paměťové náročnosti byly vytvořeny tyto pomocné algoritmy:

- **Vzorkování dat** - z datasetu je vybrán reprezentativní vzorek, který se do operační paměti vejde. Tyto algoritmy pracují s určitou mírou přesnosti, protože mohou nastat chyby. Pravidla mohou mít support dostatečný ve vzorku, ale v celém datasetu ne. Tato situace může nastat i opačně a nedojde k vygenerování pravidel, která nemají dostatečný support ve vzorku (v celém datasetu ano).
- **Rozdělení dat** - dataset je rozdělen na menší části. Algoritmus proběhne postupně na každé z částí a výsledky jsou následně sloučeny.

3 Aktuální state of the art

Vzhledem ke stáří původních algoritmů existuje k dnešnímu dni spousta pokusů o jejich úpravu pro běh v paralelním a distribuovaném prostředí.

3.1 Úpravy algoritmu Apriori

Zajímavý pokus o úpravu algoritmu v paralelním prostředí provedl např. Sujith Mohan Velliyattikuzki [3]. Autor v této práci představuje myšlenku rozdělení datasetu rvnoměrně mezi všechny dostupné procesory. Samotný výpočet poté probíhá takto:

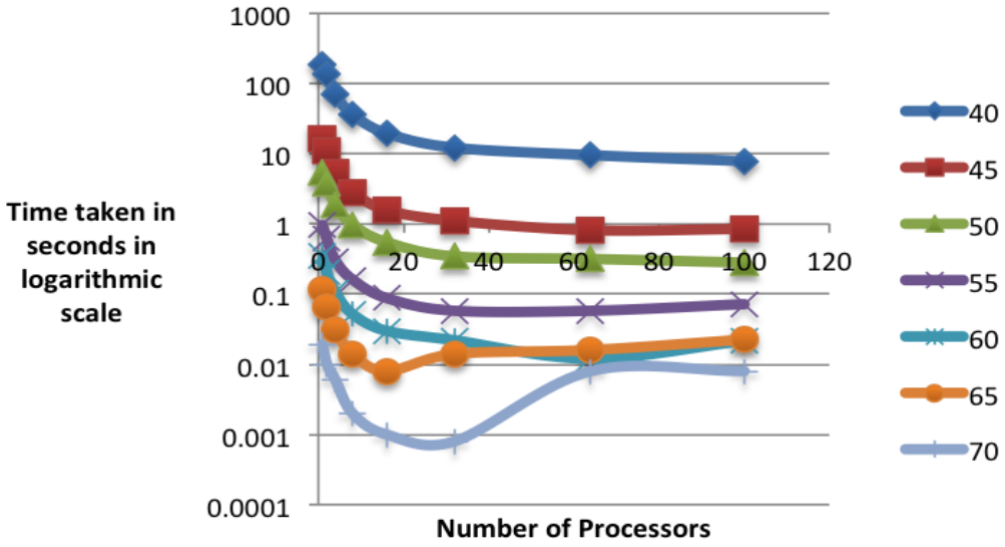
1. Hodnoty minimálního supportu a confidence jsou předány prvnímu procesoru.
2. Tyto hodnoty jsou předány všem ostatním procesorům.
3. Každý procesor vypočítá frequent itemsety velikosti 1.
4. Rozdělení datasetu mezi procesory na základě předchozího výpočtu.
5. Každý procesor vypočítá frequent itemsety na své části dat.
6. Procesor předá vypočítané lokální hodnoty globálním počítadlům.
7. Pokračování na další úroveň frequent itemsetů.
8. Opakování, dokud nejsou vygenerovány všechny frequent itemsety.

Z grafu na obr. 2 lze pozorovat, že opravdu dochází ke zkrácení doby výpočtu pomocí zvyšování počtu procesorů. Také je zde možné vidět, že od určitého počtu procesorů je již doba běhu konstantní a přidávání dalších nemá téměř žádný efekt na dobu výpočtu. Tento experiment probíhal pro hodnoty minimálního supportu mezi 40%-70% a minimální confidence 50%. Implementace byla provedena pomocí C++ a MPI.

3.2 Úpravy algoritmu Eclat

Úpravu algoritmu provedl navhrnul např. Mohammed J. Zaki [4]. Tento algoritmus je založen na výpočtu supportu pomocí průniku transakcí náležícím příslušným položkám. Tyto transakce můžeme číst současně více procesy, což výrazně ulehčí paralelizaci. Úpravu založenou na této myšlence navrhnul pan Zaki takto:

1. Rozdělit dataset rovnými díly mezi všechna vlákna.
2. Vygenerovat odděleně lokální support pro itemsety velikosti 1 a 2 v rámci každé části.
3. Sloučení lokálních počítadel supportu do globálních a vygenerování všech frequent itemsetů velikosti 1 a 2.



Obrázek 2: Velliyattikuzhi - zrychlení paralelního Apriori [3]

4. Rozdělit frequent itemsety velikosti 2 do tříd ekvivalence podle prefixů.
5. Rozdělit vygenerované třídy ekvivalence mezi procesy.
6. Každý proces si vygeneruje vertikální seznamy transakcí pro všechny frequent itemsety velikosti 2.
7. Všechny procesy si navzájem vymění informace o lokálních seznamech transakcí a vygenerují globální seznamy
8. Rekurzivně vygenerovat všechny frequent itemsety pomocí průniku itemsetů sdílejících stejnou třídu ekvivalence.

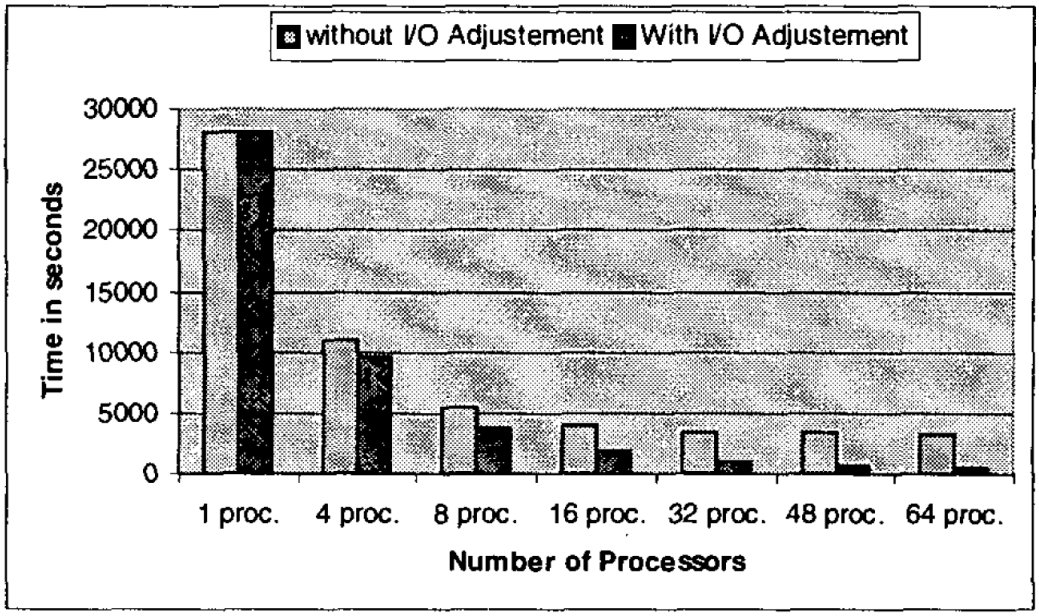
Velkou výhodou tohoto přístupu je minimální potřeba synchronizace. Většinu výpočtu běží procesy asynchronně a nezávisle na sobě, kromě generování globálních seznamů transakcí.

3.3 Úpravy algoritmu FP Growth

Jednou z možných úprav algoritmu FP Growth pro paralelní běh v prostředí big data provedli O.R. Zaiane, M. El-Hajj a P.Lu [5], kteří navrhli algoritmus MLFPT - Multiple Local Frequent Pattern Trees. Tento algoritmus je založen na myšlence konstrukce oddělených FP stromů pro každý procesor zvlášť. Problém synchronizace globálních počítadel supportu je zde vyřešen pomocí propojení počítadel lokálních.

Algoritmus se skládá ze dvou fází:

1. **Vygenerování stromů** - dataset je rozdělen na rovnoměrné části mezi procesory, které vypočítají lokální support itemsetů první úrovně a v sekvenční fázi provedou vyhodnocení



Obrázek 3: Zaiane a spol. - zrychlení paralelního FP Growth [5]

do globální header table. Poté dojde k vytvoření fp stromu pro každý procesor z přidělené části transakcí.

2. **Paralelní generování frequent itemsetů** - Stromy jsou nyní nasdílené pro všechny procesory a dojde k výpočtu rovnoměrnému rozdělení položek z header table mezi procesory. Poté dojde ke sloučení výsledků.

Ověření účinku paralelizace bylo provedeno na datasetech o rozsahu 1 - 50 milionů transakcí. Přínos tohoto řešení byl jasně zřetelný na obr. 3

3.4 Další možnosti úprav

Jednou z dalších úprav je úprava apriori za použití frameworku Map Reduce pro Hadoop od Brijendry Singha [6]. Tato práce vychází z poznatku, že nejdéle trvá výpočet pro kandidáty velikosti 2, kterých je vždy nejvíce ze všech jednotlivých úrovní velikostí. Framework využili takto:

1. **Funkce Map()** - vstupem této funkce je ve formátu dvojice klíč/hodnota, kde klíč je hodnota bitového posunu od začátku datasetu k položce obsažené v hodnotě
2. **Funkce Reduce()** - výpočet supportu a filtrace položek podmínku nesplňujících

V rámci této práce byla snížena doba zpracování kandidátů velikosti 2 o více než polovinu.

4 Popis zvolených algoritmů

Pro implementaci jsem zvolil algoritmy Eclat a FP Growth, které jsou v této kapitole popsány. Obsahuje také popis algoritmu Apriori, který tvoří základ algoritmu Eclat. Příklady v této kapitole a jejich ilustrace pocházejí z [1].

4.1 Apriori

Obr. 4 znázorňuje pseudokód algoritmu Apriori. Vstupem algoritmu je binární dataset a uživatelem definovaná hodnota minimálního supportu. Algoritmus začíná vložením kandidátů velikosti 1. Cyklus while na řádcích 5-11 vypočítá support pro aktuální úroveň kandidátů pomocí funkce ComputeSupport. Tato funkce vygeneruje všechny možné podmnožiny transakcí a zvýší support všech v nich nalezených kandidátů. Poté provedeme na řádku 8 kontrolu splnění podmínky minimálního supportu a nevyhovující kandidáty na řádku 9 odstraníme. Z kandidátů podmínku splňujících jsou následně vygenerování kandidáti vyšší úrovně. Funkce ExtendPrefixTree rozšiřuje prexif pro generaci kandidátů vyšší úrovně. Když již není možné itemset rozšířit na vyšší úroveň, tak algoritmus končí.

Obrázek 5 zobrazuje kandidáty, které je možno vygenerovat z datasetu v tabulce 1. Minimální support si definujeme 50%. To v tomto případě znamená, že pro splnění se musí itemset vyskytovat v minimálně 3 transakcích. Na obr. 5 je zobrazen strom, který získáme postupnou generací kandidátů po jednotlivých úrovních. Uzly tohoto stromu jsou spojeny, pokud jejich potomci obsahují stejný prefix. Na tomto obrázku lze vidět sílu tohoto algoritmu spočívající ve vyřazení kandidátů AC a CD. Algoritmus BruteForce by z nich i nadále generoval kandidáty vyšší úrovně, které by nakonec stejně vyřadil.

Při konstrukci tohoho stromu začínáme kandidáty první úrovně. Všichni kandidáti úrovně 1 podmínku minimálního supportu splní (obr. 6). Během rozšiřování generujeme všechny možné kombinace těchto kandidátů, protože sdílejí společný prefix, jímž je prázdná množina. Vytvoříme nový strom pro kandidáty velikosti 2, na obr. 7. Položka E je odstraněna. Po výpočtu supportu dojde dojde k odstranění AC(2) a CD(2). Kandidát BCD je odstraněn proto, že obsahuje již vyřazenou podmnožinu CD(2). Všichni kandidáti třetí úrovně splňují podmínku minimálního supportu (obr. 8). Čtvrtá úroveň již umožňuje vygenerovat pouze jednoho kandidáta ABDE (obr. 9), složeného z ABD a ABE. Algoritmus po tomto kroku skončí, protože již není možné rozšířit kandidáty na pátou úroveň.

Výpočetní složitost v nejhorším případě je $O(|I| \cdot |D| \cdot 2^{|I|})$. V tomto případě je I seznam všech položek a všechny položky z tohoto seznamu splní podmínku minimálního supportu. V praxi je však tato složitost díky vyřazování kandidátů vyýrazně nižší. Bude potřeba pouze k úplných průchodů datasetem, kde k je délka množiny kandidátů nejvyšší úrovně.

APRIORI

APRIORI (D, $\mathcal{I}$, *minsup*):

$\mathcal{F} \leftarrow \emptyset$
$\mathcal{C}^{(1)} \leftarrow \{\emptyset\}$ `// Initial prefix tree with single items`
foreach $i \in \mathcal{I}$ **do** Add i as child of $\emptyset$ in $\mathcal{C}^{(1)}$ with $sup(i) \leftarrow 0$
$k \leftarrow 1$ `// k denotes the level`
while $\mathcal{C}^{(k)} \neq \emptyset$ **do**
    COMPUTESUPPORT ($\mathcal{C}^{(k)}$, **D**)
    foreach *leaf* $X \in \mathcal{C}^{(k)}$ **do**
        if $sup(X) \geq minsup$ **then** $\mathcal{F} \leftarrow \mathcal{F} \cup \{(X, sup(X))\}$
        else remove X from $\mathcal{C}^{(k)}$
    $\mathcal{C}^{(k+1)} \leftarrow$ EXTENDPREFIXTREE ($\mathcal{C}^{(k)}$)
    $k \leftarrow k+1$
return $\mathcal{F}^{(k)}$

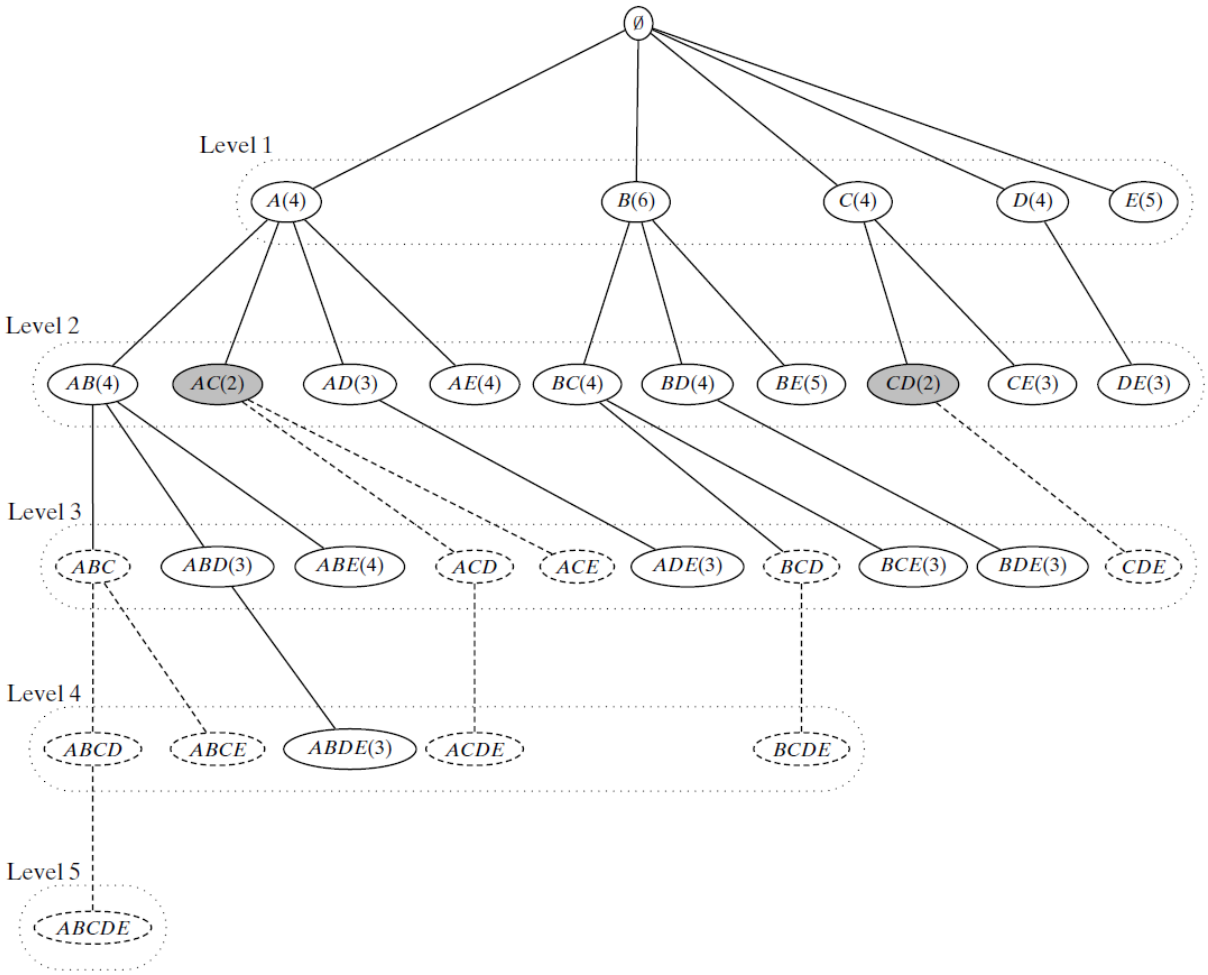
COMPUTESUPPORT ($\mathcal{C}^{(k)}$, D):

foreach $\langle t, \mathbf{i}(t) \rangle \in \mathbf{D}$ **do**
    foreach k-*subset* $X \subseteq \mathbf{i}(t)$ **do**
        if $X \in \mathcal{C}^{(k)}$ **then** $sup(X) \leftarrow sup(X) + 1$

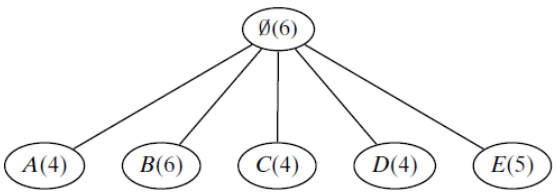
EXTENDPREFIXTREE ($\mathcal{C}^{(k)}$):

foreach *leaf* $X_a \in \mathcal{C}^{(k)}$ **do**
    foreach *leaf* $X_b \in$ SIBLING(X_a), *such that* $b > a$ **do**
        $X_{ab} \leftarrow X_a \cup X_b$
        `// prune candidate if there are any infrequent subsets`
        if $X_j \in \mathcal{C}^{(k)}$, ***for all*** $X_j \subset X_{ab}$, *such that* $|X_j| = |X_{ab}| - 1$ **then**
            Add X_{ab} as child of X_a with $sup(X_{ab}) \leftarrow 0$
    if *no extensions from* X_a **then**
        remove X_a, and all ancestors of X_a with no extensions, from $\mathcal{C}^{(k)}$
return $\mathcal{C}^{(k)}$

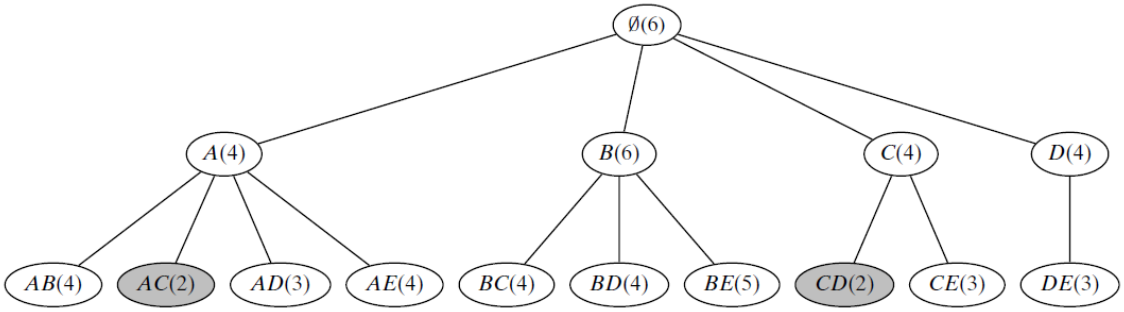
Obrázek 4: Pseudokód Apriori [1]



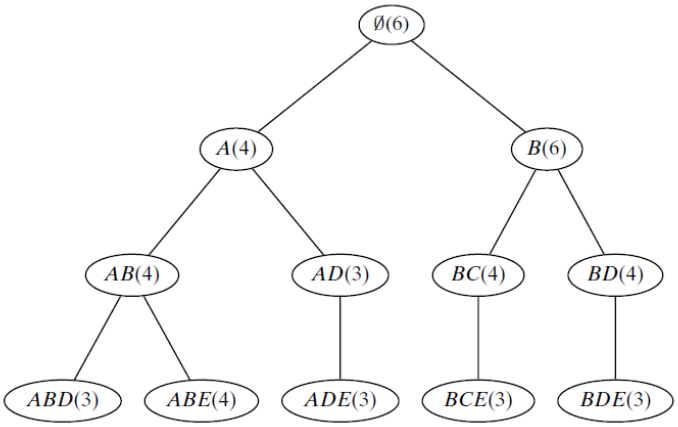
Obrázek 5: Apriori - strom [1]



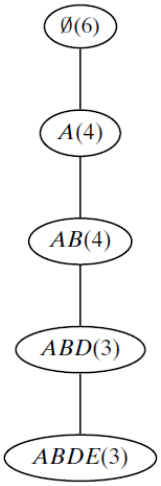
Obrázek 6: Apriori - kandidáti úrovně 1 [1]



Obrázek 7: Apriori - kandidáti úrovně 2 [1]



Obrázek 8: Apriori - kandidáti úrovně 3 [1]



Obrázek 9: Apriori - kandidáti úrovně 4 [1]

4.2 Eclat

Výpočet supportu pro jednotlivé kandidáty lze výrazně urychlit vhodnější reprezentací datasetu. Algoritmus eclat je založen na Apriori, používá však vertikální dataset, který umožní mnohem rychlejší výpočty. U apriori bylo nutné vygenerovat všechny kandidáty každé transakce a kontrolovat, zda existují v prefixovém stromu. To může vést ke generaci spousty kandidátů, kteří v prefixovém stromu ani neexistují a ověřování jejich supportu je zbytečné.

Hlavní výhodou tohoto přístupu je možnost výpočtu supportu pomocí průniku sloupců. Pokud např. potřebujeme vypočítat support itemsetu složeného z pěti položek, tak stačí spočítat průnik sloupců odpovídajících příslušným položkám. Tím získáme identifikátory transakcí, které obsahují všechny položky itemsetu a počet těchto identifikátorů reprezentuje hodnotu supportu.

Pokud víme, že seznam transakcí pro položku A je 1345 a pro položku C je 2456, tak můžeme vypočítat support itemsetu AC pomocí průniku A∧B = 1345∧2456 = 45.

Pseudokód algoritmu Eclat je na obr. 10. Přepokládáme, že P obsahuje pouze frequent itemsety, což je na začátku prázdná množina. Poté postupujeme tak, že pro každý frequent itemset patřící do P postupně spočítáme kardinalitu průniku se všemi ostatními itemsety z P. Pokud dojde ke splnění podmínky minimálního supportu, tak je daný průnik přidán do nové úrovně třídy P, která obsahuje X_a jako prefix. Následné rekurzivní volání algoritmu nalezne všechny možnosti rozšíření X_a. Toto se opakuje, dokud již žádné další rozšíření není možné.

Pro demonstraci běhu algoritmu použijeme dataset z tabulky 3. Grafické zobrazení je na obr. 11. Nastavení supportu použijeme opět 50%, tedy 3 transakce. Výchozí třída prefixů je:

$$\text{P} = \{\text{<A,1345>,<B,123456>,<C,2456>,<D,1356>,<E,12345>}\}$$

Nyní provedeme průnik transakcí prvního prvku A se všemi osatními (B,C,D,E) a získáme hodnoty supportu pro AB, AC, AD a AE. Kontrolou minimálního supportu neprojde AC a je tedy vyřazeno (na obrázku označeno šedě). Z ostatních itemsetů je vygenerována nová třída prefixů:

$$\text{P} = \{\text{<AB,1345>,<AD,135>,<AE,1345>}\}$$

Tuto třídu opět rekurzivně rozšíříme o průniky s B,C,D a E, čímž získáme třídu prefixů:

$$\text{P} = \{\text{<BC,2456>,<BD,1356>,<BE,12345>}\}$$

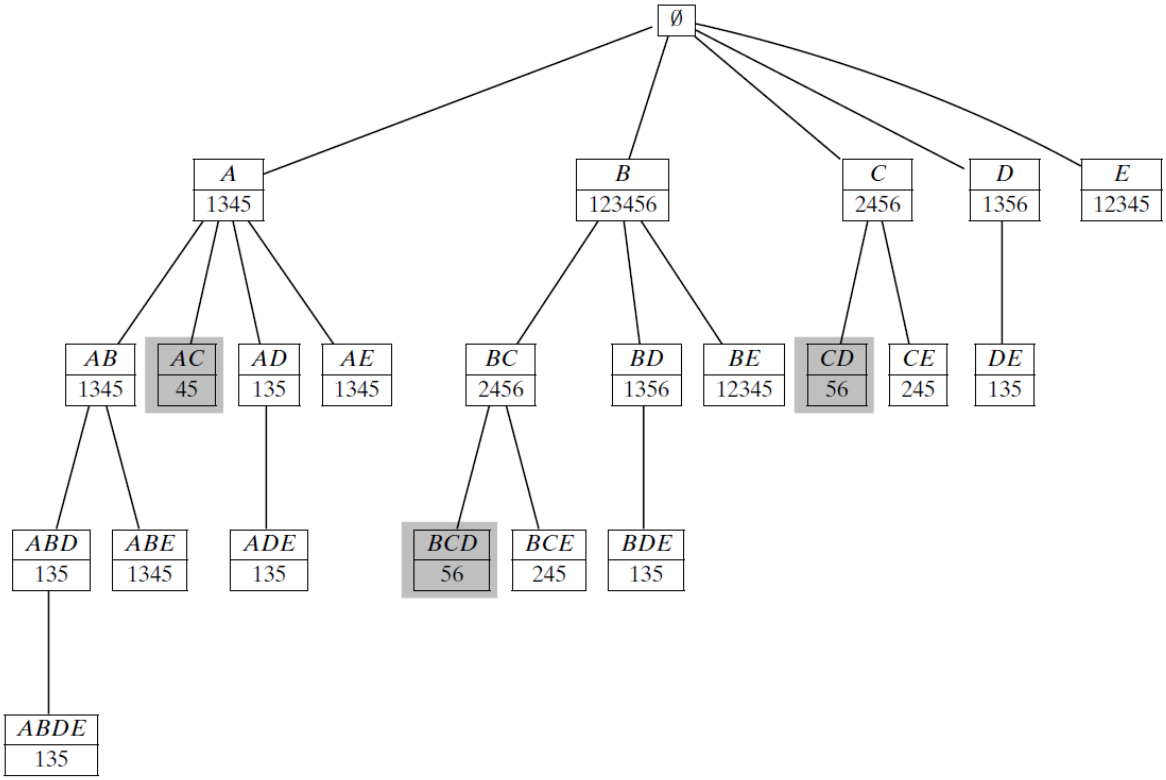
Pro ostatní větve se pokračuje obdobně dle obrázku.

Výpočetní složitost v nejhorším případě je $O(|D|\cdot 2^{|I|})$, protože opět můžou všechny položky splnit minimální support a průnik transakcí dvou položek je maximálně $O(|D|)$.

ECLAT

```
// Initial Call: F ← ∅, P ← {⟨i, t(i)⟩ | i ∈ I, |t(i)| ≥ minsup}
ECLAT (P, minsup, F):
1 foreach ⟨X_a, t(X_a)⟩ ∈ P do
2     F ← F ∪ {(X_a, sup(X_a))}
3     P_a ← ∅
4     foreach ⟨X_b, t(X_b)⟩ ∈ P, with X_b > X_a do
5         X_ab = X_a ∪ X_b
6         t(X_ab) = t(X_a) ∩ t(X_b)
7         if sup(X_ab) ≥ minsup then
8             P_a ← P_a ∪ {⟨X_ab, t(X_ab)⟩}
9     if P_a ≠ ∅ then ECLAT (P_a, minsup, F)
```

Obrázek 10: Pseudokód Eclat [1]



Obrázek 11: Eclat [1]

4.3 FP Growth

Algoritmus FP Growth nahrazuje generaci kandidátů použitím datové struktury frequent pattern tree (FP strom). Každý uzel tohoto stromu obsahuje identifikátor položky zboží, hodnotu supportu, odkaz na rodičovský uzel a odkazy na uzly potomků.

Konstrukce stromu začíná vytvořením kořenového uzlu, který obsahuje položku ∅. Poté do stromu postupně vložíme všechny transakce datasetu. Pokud transakce sdílí prefix s nějakou již dříve vloženou, tak u již existujících uzlů pouze zvyšujeme hodnotu supportu. Pro zbývající položky vytvoříme nové uzly s hodnotou supportu 1. Konstrukce končí po vložení poslední transakce.

Pro zajištění co největší kompaktnosti je nutné umístit nejčastěji se vyskytující položky co nejblíže ke kořeni FP stromu. Toho dosáhneme sestupným seřazením položek obsažených v datasetu ještě před samotným generováním stromu. Napřed vypočítáme support pro jednoprvkové množiny položek. Poté odstraníme množiny nesplňující minimální support a utřídíme položky sestupně podle vypočítaného supportu. Každou transakci nyní před vložením do stromu setřídíme podle tohoto pořadí.

V datasetu z tabulky 2 získáme utříděním seznam B(6), E(5), A(4), C(4), D(4). Transakce utřídíme podle tohoto seznamu, takže např. z ABDE se stane BEAD. Postup vložení všech transakcí znázorňují obr. 13, 14, 15, 16, 17 a 18.

Po úspěšné konstrukci stromu můžeme generovat frequent itemsety přímo z tohoto stromu a původní dataset již nepotřebujeme. Pseudokód pro generaci frequent itemsetů z FP stromu je na obr. 12. Algoritmus přijímá na vstupu FP strom R vytvořený ze vstupního datasetu D a aktuální prefix, který je na začátku prázdný.

Pro každou položku je vygenerován příslušný podmíněný FP strom vzestupně podle hodnoty supportu. Z každého výskytu položky je vygenerována cesta do kořene stromu (řádek 13). Hodnota supportu dané položky na konci cesty je zaznamenána do proměnné cnt (řádek 14) a cesta je vložena do nového podmíněného stromu R_x, kde X množina získaná rozšířením aktuálního prefixu P položkou i. Při vkládání cesty postupujeme stejně jako při generování základního stromu. Poté voláme algoritmus FP Growth rekurzivně s podmíněným stromem R_x a novým prefixem X (řádek 16). Rekurze skončí když vstupní strom má pouze jednu cestu. Tento strom je poté zpracován vygenerováním všech množin kombinací prvků v něm obsažených a support každé množiny odpovídá nejnižšímu supportu z obsažených prvků (řádky 2 až 6).

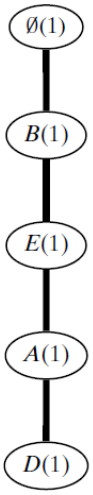
Pro demonstraci běhu algoritmu použijeme strom z předcházejícího příkladu (obr. 18). Minimální support bude opět 50% (3). Úvodní prefix je P=∅. a množina položek splňujících minimální support je B(6), E(5), A(4), C(4) a D(4). Poté pro tyto položky vytvoříme podmíněné stromy. Strom pro položku D je na obr. 21. V základním stromu (obr. 18) existují tři cesty z kořene k položce D:

<BCD, sup = 1; BEACD, sup = 1; BEAD, sup = 2>

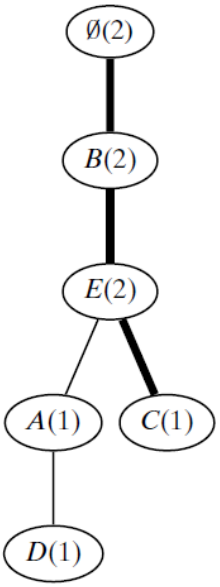
FPGROWTH

```
// Initial Call: R ← FP-tree(D), P ← ∅, F ← ∅
FPGROWTH (R, P, F, minsup):
1  Remove infrequent items from R
2  if ISPATH(R) then // insert subsets of R into F
3      foreach Y ⊆ R do
4          X ← P ∪ Y
5          sup(X) ← min_{x∈Y}{cnt(x)}
6          F ← F ∪ {(X, sup(X))}
7  else   // process projected FP-trees for each frequent item i
8      foreach i ∈ R in increasing order of sup(i) do
9          X ← P ∪ {i}
10         sup(X) ← sup(i) // sum of cnt(i) for all nodes labeled i
11         F ← F ∪ {(X, sup(X))}
12         R_X ← ∅ // projected FP-tree for X
13         foreach path ∈ PATHFROMROOT(i) do
14             cnt(i) ← count of i in path
15             Insert path, excluding i, into FP-tree R_X with count cnt(i)
16         if R_X ≠ ∅ then FPGROWTH (R_X, X, F, minsup)
```

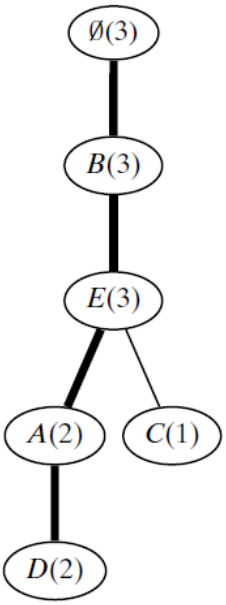
Obrázek 12: Pseudokód FP Growth [1]



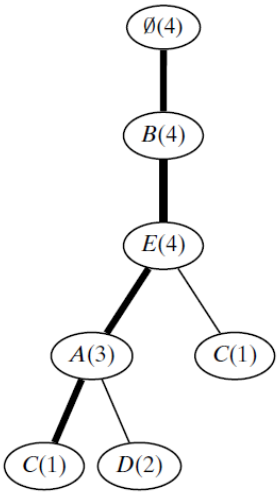
Obrázek 13: FP Growth, vložení (1, BEAD) [1]



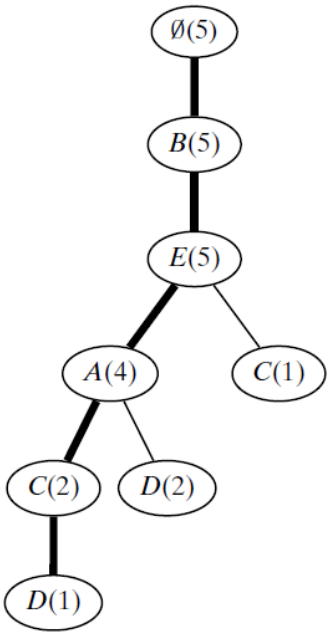
Obrázek 14: FP Growth, vložení (2, BEC) [1]



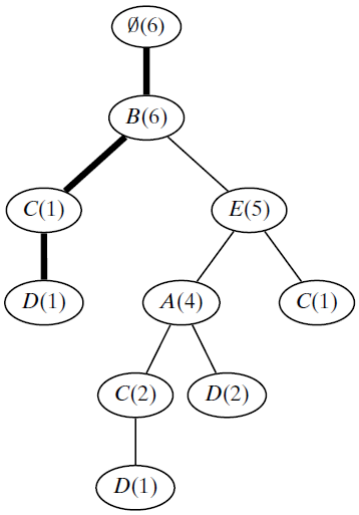
Obrázek 15: FP Growth, vložení (3, BEAD) [1]



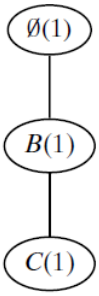
Obrázek 16: FP Growth, vložení (4, BEAC) [1]



Obrázek 17: FP Growth, vložení (5, BEACD) [1]



Obrázek 18: FP Growth, vložení (6, BCD) [1]



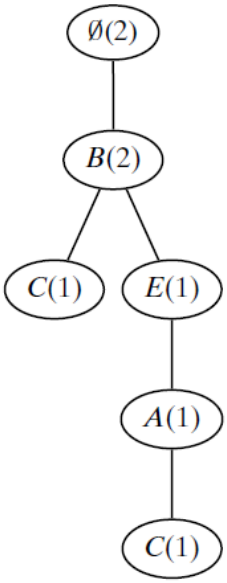
Obrázek 19: FP Growth, vložení BC [1]

Tyto tři cesty jsou postupně vloženy do podmíněného FP stromu R_d. Postup vkládání zobrazují obr. 19, 20 a 21. Výsledný strom je opět zpracován rekurzivně.

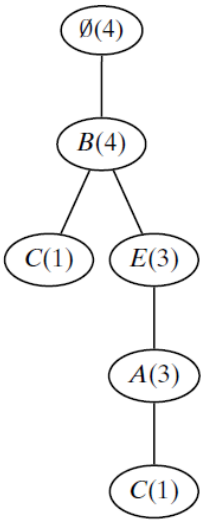
Při zpracování stromu R_d se stává prefixem $P = D$. Po odstranění položky C, která nesplní minimální support získáme strom s jednou cestou B(4)-E(3)-A(3). Vygenerujeme všechny možné množiny kombinací a doplníme je o D. Získáme frequent itemsety DB(4), DE(3), DA(3), DBE(3), DBA(3), DEA(3) a DBEA(3). Nyní přecházíme na další položku hlavního stromu. Podmíněné stromy pro C,A i E mají je jednu cestu, což umožní přímo vygenerovat frequent itemsety {CB(4), CE(3), CBE(3)}, {AE(4), AB(4), AEB(4)} a EB(5). Tento proces je zobrazen na obr. 22.

4.4 Generování pravidel

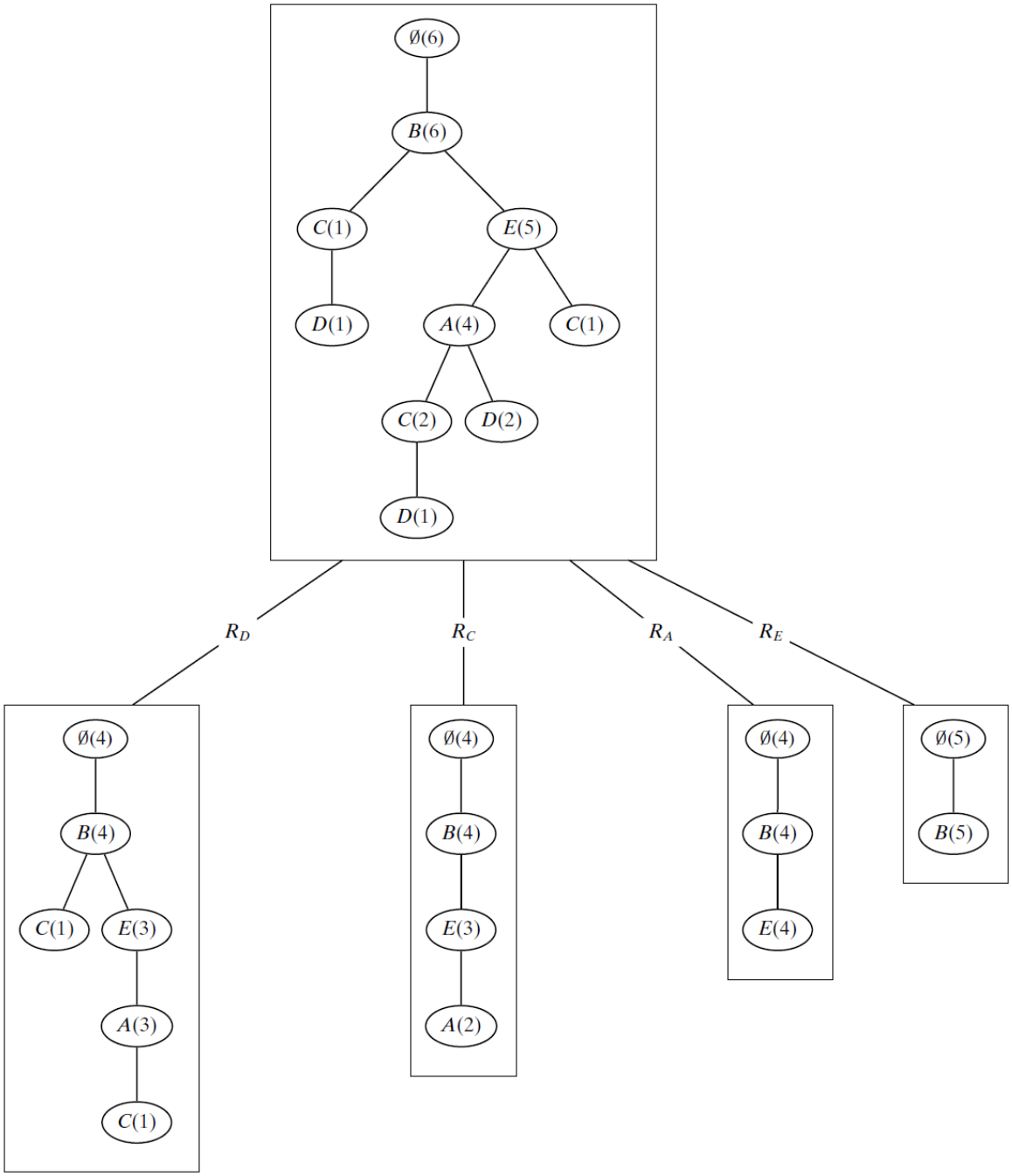
Výše zmíněné algoritmy se postarají o první fázi procesu - ověření supportu. Poté přichází na řadu algoritmus pro vygenerování pravidel a ověření confidence. Nechť Z je frequent itemset se všech vygenerovaných F. Cílem je vygenerovat pro každý itemset Z z F všechna pravidla a



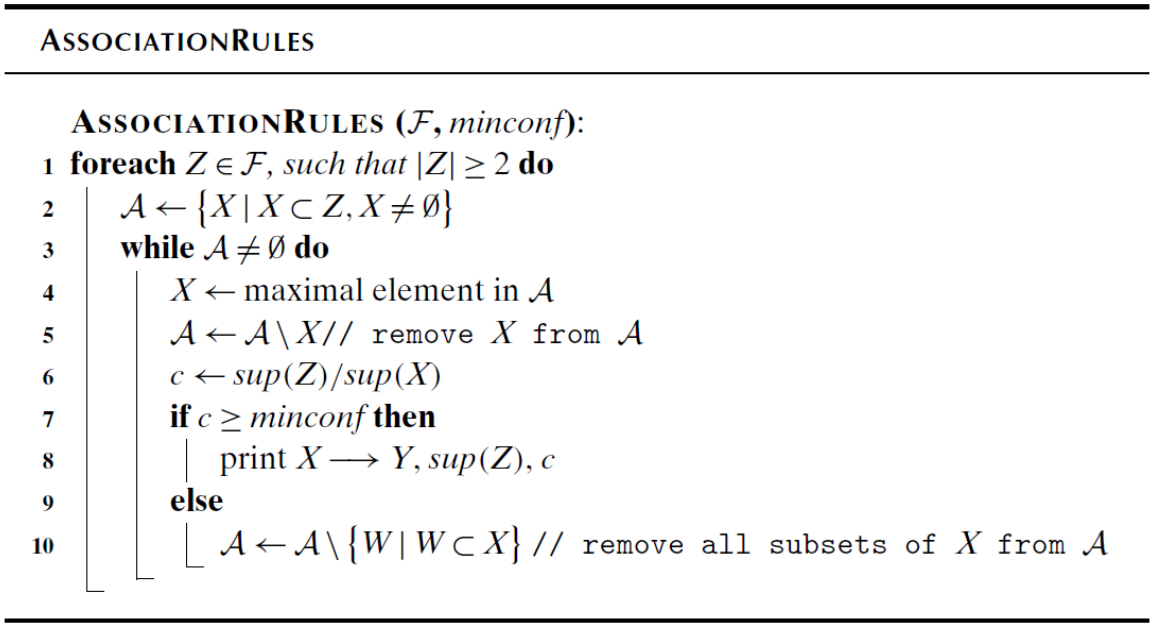
Obrázek 20: FP Growth, vložení BEAC [1]



Obrázek 21: FP Growth, vložení BEA [1]



Obrázek 22: FP Growth - generování podstromů [1]



Obrázek 23: Pseudokód algoritmu generování asociačních pravidel [1]

ověřit jejich confidence. Pro všechny množiny X a Y, jenž jsou podmnožinami Z vypočítáme confidence:

$$c = \frac{sup(X \cup Y)}{sup(X)} = \frac{sup(Z)}{sup(X)}$$

Pokud confidence c splňuje podmínku minimální confidence, tak je pravidlo považováno za **silné**.

Na obr. 23 je pseudokód algoritmu pro generování asociačních pravidel z frequent itemsetů o minimální velikosti 2. Na řádku 2 inicializujeme seznam všech podmnožin aktuálního frequent itemsetu Z. Na řádku 7 pro každé X z A ověřujeme zda pravidlo splní podmínku minimální confidence.

Pro ukázku na příkladu použijeme frequent itemset ABDE(3) z datasetu v tabulce 1. Minimální confidence bude 0.9. Vygenerujeme podmnožiny:

{ABD(3),ABE(4),ADE(3),BDE(3),AB(3),AD(4),AE(4),BD(4),BE(5),DE(3)}

První podmnožinou bude X = ABD a confidence pravidla ABD→E je 3/3 = 1. Následuje podmnožina X = ABE, ale pravidlo ABE→D není silné, protože conf(ABD→D) = 3/4 = 0.75. Proto můžeme odstranit všechny podmnožiny ABE, což znamená že získáme:

{ADE(3),BDE(3),AD(4),BD(4),DE(3)}

Dalším kolem projdou všechny podmnožiny kromě X = BD, protože conf(BD→AE) = 3/4 = 0.75, což vede k odstranění všech podmnožin BD a získáme:

{DE(3)}

Finální kolekce vygenerovaných asociačních pravidel bude:

ABD→E (conf 1)
ADE→B (conf 1)
BDE→A (conf 1)
AD→BE (conf 1)
DE→AB (conf 1)

5 Implementace zvolených algoritmů

Pro implementaci jsem zvolil algoritmy Eclat a FP Growth a implementoval je jako konzolovou aplikaci v programovacím jazyce C++ za použití Microsoft Visual Studio 2015.

Výsledný program se skládá z těchto souborů:

- **projekt.cpp** - hlavní soubor obsahující funkci main()
- **eclat.h** - hlavičkový soubor obsahující deklarace funkcí pro algoritmus Eclat
- **fpgrowth.h** - hlavičkový soubor obsahující deklarace funkcí pro algoritmus FP Growth
- **rulesGeneration.h** - hlavičkový soubor obsahující deklarace funkcí pro generaci asociačních pravidel
- **eclat.cpp** - hlavičkový soubor obsahující definice funkcí pro algoritmus Eclat
- **fpgrowth.cpp** - hlavičkový soubor obsahující definice funkcí pro algoritmus FP Growth
- **rulesGeneration.cpp** - hlavičkový soubor obsahující definice funkcí pro generaci asociačních pravidel a obslužných metod pro práci s datasetem

5.1 Hlavní program

Hlavní řídící program se nachází ve funkci main() souboru projekt.cpp. Tato funkce na začátku načítá vstupní parametry ze souboru settings.txt.

Program načítá hodnoty těchto parametrů:

- **minimální support** - double - např. 0.1
- **minimální confidence** - double - např. 0.9
- **počet vláken** - int - počet vláken, ve kterém má běžet paralelní část programu, např. 4
- **vstupní soubor** - string - cesta a název vstupního CSV souboru datasetu, např. C:/Dataset.csv
- **výstupní soubor** - string - cesta a název výstupního textového souboru pro zápis pravidel, např. Output.txt
- **metoda** - int - volba použitého algoritmu: 0 = Eclat, 1 = FP Growth

Po načtení vstupních parametrů postupuje program takto:

1. Načtení všech názvů zboží do pomocné proměnné attributes, která zajistí uchování pro pozdější generování pravidel.
2. Vygenerování frequent itemsetů pomocí algoritmu Eclat nebo FPgrowth.

3. Vygenerování asociačních pravidel z vygenerovaných frequent itemsetů.

4. Zápis vygenerovaných pravidel do výstupního souboru.

5.2 Implementace algoritmu Eclat

Algoritmus Eclat se nachází v souborech eclat.h a eclat.cpp.

5.2.1 Hlavní funkce

Běh celého algoritmu zastřešuje jedna hlavní funkce *eclat*, která zajišťuje volání všech ostatních funkcí. Tato funkce na základě svého vstupního parametru načte dataset ze vstupního souboru pomocí funkce *loadData*. Poté provede konverzi načteného binárního datasetu na vertikální pomocí funkce *convertToVertical*. Následně předá vertikální dataset funkci *generateFrequentItems*, od které přijme vygenerované frequent itemsety, které jsou návratovou hodnotou této funkce.

5.2.2 Funkce generateFrequentItemsets

V této funkci dojde napřed k vygenerování frequent itemsetů velikosti 1 a 2 pomocí speciální funkce *generateL2*. Tato funkce napřed odfiltruje jednotlivé položky, které nesplní support a ze zbytku vygeneruje všechny možné dvojice.

Následná generace frequent itemsetů vyšších úrovní probírá pomocí těchto funkcí:

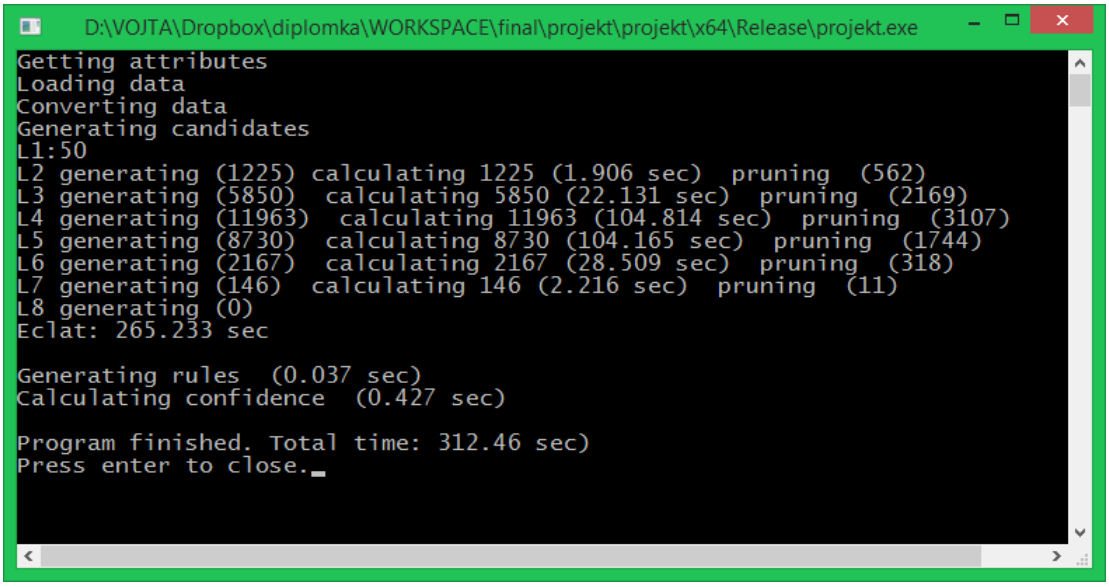
1. generateTuples - vygenerování všech kandidátů o 1 vyšší úrovně s důrazem na generování pouze platných kandidátů bez duplicit přesných či s lexikografickým rozdílem

2. calculateSupport - výpočet supportu pomocí průniků

3. prune - odstranění všech kandidátů nesplňujících minimální support

Tento proces se opakuje, dokud již není možné vygenerovat nové kandidáty vyšší úrovně. Ukázka běhu programu pro algoritmus Eclat je na obr. 24. Ukázka výpočtu supportu ve funkci *calculateSupport* je ve výpisu 1.

```
int i;

#pragma omp parallel for num_threads(p_threads)
   for (i = 0; i < p_temp_tuples.size(); i++)
   {
      vector<int> temp_vec;
      set_intersection(p_dataset[p_temp_tuples[i][0]].begin(), p_dataset[
         p_temp_tuples[i][0]].end(), p_dataset[p_temp_tuples[i][1]].begin(),
         p_dataset[p_temp_tuples[i][1]].end(), back_inserter(temp_vec));
```



Obrázek 24: Ukázka běhu programu pro algoritmus Eclat

```
    if (p_L > 2)
    {
       for (int l = 2; l < (p_L); l++)
       {
          vector<int> temp_vec2;
          set_intersection(temp_vec.begin(), temp_vec.end(), p_dataset[
              p_temp_tuples[i][l]].begin(), p_dataset[p_temp_tuples[i][l]].end
              (), back_inserter(temp_vec2));
          temp_vec = temp_vec2;
       }
    }
    p_temp_tuples[i][p_L] = temp_vec.size();
 }
 sort(p_temp_tuples.begin(), p_temp_tuples.end() - 1);
```

Výpis 1: Eclat - část funkce calculateSupport

5.3 Implementace algoritmu FP Growth

Algoritmus FP Growth se nachází v souborech fpgrowth.h a fpgrowth.cpp.

5.3.1 Hlavní funkce

Také tento algoritmus je zastřešen hlavní funkcí *fpgrowth*, která zajišťuje volání všech ostatních funkcí. Tato funkce na základě svého vstupního parametru načte dataset ze vstupního souboru

pomocí funkce *loadData*. Poté provede konverzi načteného binárního datasetu na transakční pomocí funkce *convertToTransactional*. Následně předá transakční dataset funkci *ConstructFPTree*, která vygeneruje FP strom. Výsledný strom je předán funkci *ProcessFPTree*, která vrátí frequent itemsety. Jejich formát je poté ještě před výstupem upraven pomocí funkce *processFrequentItemsets* aby měl stejný formát jako výstup algoritmu Eclat.

5.3.2 Funkce processFPTree

Účelem této funkce je zpracovat vygenerovaný FP strom a vygenerovat z něj všechny frequent itemsety.

Zpracování stromu probíhá takto:

1. Kontrola, zda vstupní strom není prázdný. Pokud ano, tak funkce vrátí prázdný vektor.
2. Kontrola, zda strom obsahuje jednu cestu. Pokud ano, tak dojde k vygenerování všech možných itemsetů z uzlů tohoto stromu, výpočtu supportu a vrácení těch, které splní podmínku minimálního supportu.
3. Pro všechny položky vzestupně podle supportu provést:
   (a) vygenerovat conditional pattern base pro aktuální položku
   (b) vygenerovat transakce pro aktuální conditional pattern base
   (c) vygenerovat FP strom pro transakce z předchozího kroku
   (d) rekurzivně volat funkci processFPTree dokud není výsledný strom prázdný nebo obsahuje jen jednu cestu
   (e) přidat aktuální frequent itemsety do celkového výsledku
4. Vrátit vygenerované frequent itemsety po ukončení rekurze.

Ukázka funkce ProcessFPTree je ve výpisu 2.

```
vector<pair<vector<int>, int>> ProcessFPTree(FPTree tree, double p_minsup, int
    p_num_threads, int recursion)
{
   //check if input tree is empty
   if (tree.root->childrenNodes.size() == 0) return vector<pair<vector<int>,
      int>>{};
   //check if input tree is single path
   bool singlepath = true;
   FPTreeNode * curNode = tree.root;
   while (curNode->childrenNodes.size() > 0)\{ ... \}
```

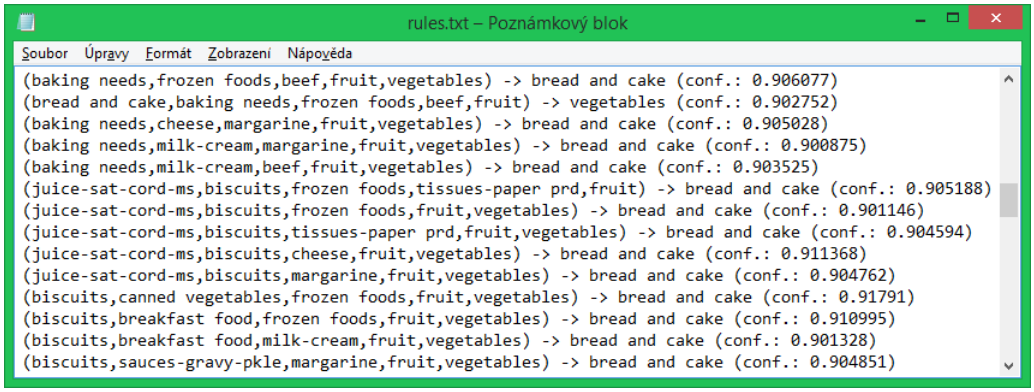
```
if (singlepath == true)\{ ... \}
else
{
   vector<pair<vector<int>, int>> current_freq_items;
   if (recursion > 0) p_num_threads = 1;
   recursion++;

   #pragma omp parallel for num_threads(p_num_threads)
   for (int i = tree.header_table.size() - 1; i >= 0; i--)
   {
      //generate conditional pattern base for actual item
      vector<pair<vector<int>, int>> conditionalPatternBase;
      //for all paths of actual item
      for (int j = 0; j < tree.header_table[i].second.size(); j++)\{ ... \}
      //generate transactions for actual conditional pattern base
      vector<vector<int>> temp_transactions;
      for (int j = 0; j < conditionalPatternBase.size(); j++)\{ ... \}
      //conditional FP tree
      FPTree conTree = ConstructFPTree(temp_transactions, p_minsup);
      //recursion until tree is empty or single path
      vector<pair<vector<int>, int>> conditional_patterns = ProcessFPTree(
         conTree, p_minsup, p_num_threads, recursion);
      ...
```

Výpis 2: FP Growth - část funkce ProcessFPTree

5.4 Implementace algoritmu generování asociačních pravidel

Funkce pro generaci asociačních pravidel se nacházejí v souborech rulesGeneration.h a rulesGeneration.cpp. Generace je zastřešena hlavní funkcí *generateRules*. Samotné generování probíhá pomocí těchto dalších funkcí:

1. *generateRulesFromFrequentItems* - tato funkce vygeneruje z každého frequent itemsetu všechna možná pravidla
2. *calculateConfidence* - tato funkce vypočítá confidence pro každé pravidlo a ponechá pouze ty, které splňují minimální confidence
3. *outputRules* - zapíše pravidla do výstupního souboru. Ukázka výstupního souboru je na obr. 25



Obrázek 25: Ukázka výstupního souboru programu

5.5 Úpravy pro běh v paralelním prostředí

Pro paralelizaci jsem použil OpenMP 2.0. Měřením doby běhu jednotlivých částí algoritmů jsem identifikoval části, které tvoří většinu doby výpočtu a paralelizace zde přinese největší zvýšení výkonu a efektivity.

5.6 Úprava algoritmu Eclat

Paralelizace byla provedena ve funkci *calculateSupport* při výpočtu supportu pro jednotlivé položky. Ten probíhá pomocí průniků identifikátorů transakcí pro zpracovávané položky a není problém, když více vláken čte současně transakce pro stejnou položku. Proto není nutné provádět žádnou synchronizaci ani předávání dat mezi vlákny. Negativním vlivem může být pouze různé pořadí výsledných itemsetů, ale ty je stejně nutné před dalším zpracováním setřídit i při sekvenčním běhu algoritmu.

5.7 Úprava algoritmu FP Growth

Paralelizace byla provedena ve funkci *ProcessFPTree* při generování položek ze stromu, který obsahuje více než jednu cestu. Paralelně probíhá zpracování různých položek z header table, které jsou na sobě nezávislé. Vzhledem k obsažené rekurzi je součástí parametrů této funkce také příznak rekurze, kterým je zajištěno nepřekročení maximálního definovaného počtu vláken.

Výsledky těchto úprav ilustují experimenty z kapitoly 6.

6 Experimenty a měření

S implementací jsem provedl více experimentů nad testovacími daty. Tato kapitola popisuje ty nejzajímavější, které ilustrují cíle celé práce. Správnost výstupních pravidel byla validována analytickým softwarem Weka[7].

6.1 Testovací HW

Experimenty byly prováděny na dvou serverech:

1. **Školní** - argexpr.vsb.cz, 2x Intel Xeon E5-2680, 768GB RAM, Windows Server 2012 R2 Standard, 20 fyzických jader (40 logických)
2. **Soukromý** - 2x Intel Xeon E5645, 96GB RAM, Windows Server 2008 R2 Enterprise,12 fyzických jader (24 logických)

6.2 Datasety

Základem všech použitých datasetů je soubor supermarket z analytického softwaru Weka[7]. Vzhledem k nedostupnosti reálných datasetů z obchodních řetězců byly větší datasety vygenerovány uměle z tohoto datasetu. Všechny datasety jsou složeny z transakcí pro 216 typů zboží.

- **Dataset 1** - 925400 transakcí
- **Dataset 2** - 4627000 transakcí
- **Dataset 3** - 46270000 transakcí

6.3 Experiment 1

Cílem tohoto experimentu bylo vyzkoušet na malém datasetu chování paralelních úprav pro různé hodnoty supportu a poté vzájemně srovnat algoritmy.

Parametry experimentu:

- **Server:** školní
- **Dataset:** dataset 1
- **Minimální support:** postupně 0.05, 0.1 a 0.2
- **Minimální confidence:** 0.9
- **Počet vláken:** 1-8

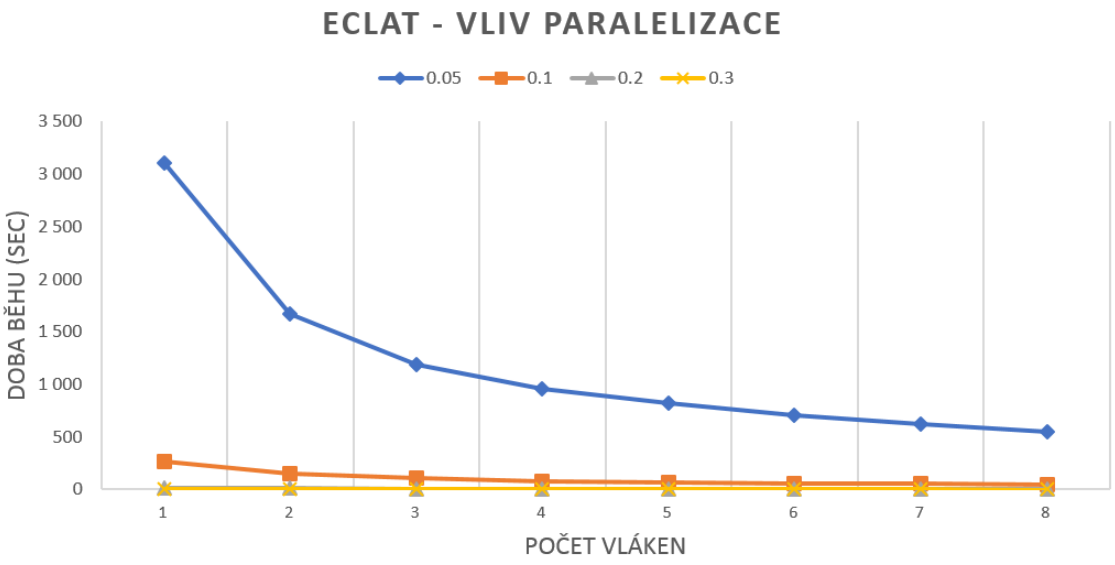
Naměřené hodnoty pro algoritmus Eclat jsou v tabulce 4 a pro FP Growth v tabulce 5. Hodnoty v této tabulce jsou průměrem 30 měření.

Tabulka 4: Experiment 1 - Eclat

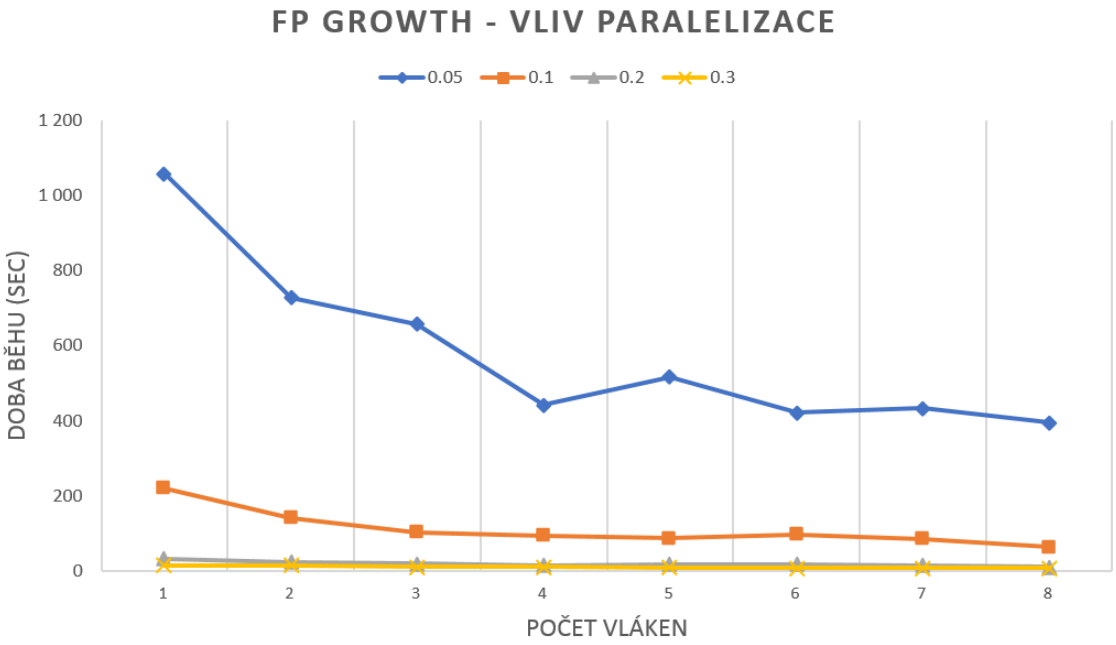
Doba běhu (sec)	**Minimální support**			
Počet vláken	**0.05**	**0.1**	**0.2**	**0.3**
1	3101	266	18	4.6
2	1665	155	10	2.86
3	1193	105	7.4	4.5
4	959	82	5.8	1.23
5	824	68	4.7	1.2
6	708	55	4.1	1.9
7	623	52	3.7	1.6
8	545	44	3.4	1.1

Tabulka 5: Experiment 1 - FP Growth

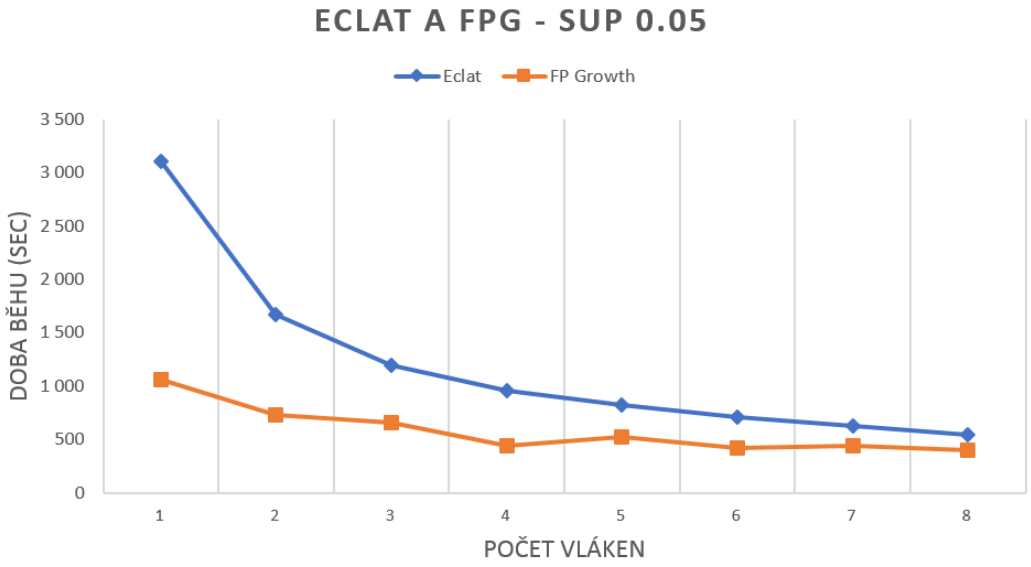
Doba běhu (sec)	**Minimální support**			
Počet vláken	**0.05**	**0.1**	**0.2**	**0.3**
1	1060	220	33	14.4
2	726	141	24	13.9
3	657	103	21	10.7
4	443	94	16	11.4
5	517	88	18.1	9.4
6	422	98	18	8.3
7	435	84	14.3	8
8	395	66	13.2	7.8



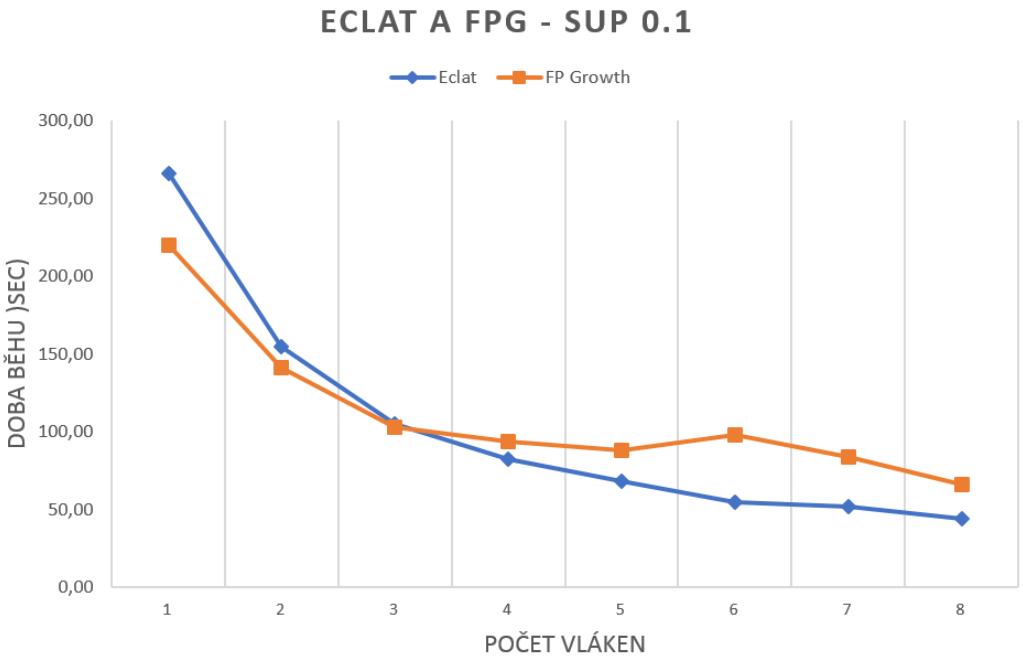
Obrázek 26: Experiment 1 - Vliv paralelizace u algoritmu Eclat pro různé hodnoty supportu



Obrázek 27: Experiment 1 - Vliv paralelizace u algoritmu FP Growth pro různé hodnoty supportu



Obrázek 28: Experiment 1 - Porovnání paralelizace Eclat vs FP Growth - support 0.05



Obrázek 29: Experiment 1 - Porovnání paralelizace Eclat vs FP Growth - support 0.1

6.3.1 Vyhodnocení experimentu 1

Vliv paralelizace pro algoritmus Eclat je v grafu na obr. 26. Z grafu i tabulky je patrný účinek paralelizace. Doba běhu se s každým zdvojnásobením počtu vláken zmenší téměř na polovinu. Minimální rozdíl mezi hodnotami supportu 0.2 a 0.3 byl způsoben tím, že při této hodnotě supportu již nebyly vygenerovány žádné frequent itemsety. Tyto hodnoty supportu proto byly z dalších experimentů vyřazeny.

Vliv paralelizace pro algoritmus Eclat je v grafu na obr. 27. Naměřené hodnoty jsou v tabulce 5. Z grafu i tabulky je patrý účinnek paralelizace. Doba běhu však neklesá lineárně jako u Eclatu. Detailním prověřením běhu algoritmu bylo zjištěno, že výkyvy nastávají kvůli jednomu vláknu, které běží výrazně déle než ostatní vlákna.

Porovnání obou algoritmů pro hodnotu supportu 0.05 je v grafu na obr. 28. Z tohoto grafu je patrné, že při nížší hodnotě supportu má algoritmus FP Growth při nižším počtu vláken výrazně vyšší výkon než Eclat. Tento rozdíl se však s přidáváním dalších vláken postupně zmenšuje.

Tyto algoritmy byly porovnány také pro hodnotu supportu 0.1, výsledný graf je na obr. 29. Zde se výkon obou algoritmů již tolik nelišil a při třech vláknech běžely oba stejně dlouhou dobu. Při dalším zvyšování vláken měl vyšší výkon algoritmus Eclat.

6.4 Experiment 2

Cílem tohoto experimentu bylo vyzkoušet na malém datasetu chování paralelních úprav při zvyšujícím se počtu vláken.

Tabulka 6: Experiment 2

Doba běhu (sec)	**Minimální support**			
	0.05		**0.1**	
Počet vláken	**Eclat**	**FPG**	**Eclat**	**FPG**
1	5563	1513	497	282
2	3005	1048	266	173
3	2083	996	186	125
4	1601	618	141	116
5	1319	718	117	110
6	1221	621	102	126
7	1105	645	95	103
8	994	587	86	90
9	932	486	80	98
10	876	489	75	95
11	825	512	70	78
12	769	436	65	78
13	716	416	61	75
14	675	470	58	73
15	641	462	56	75
16	619	456	53	72
17	602	490	52	69
18	595	476	52	73
19	589	390	51	67
20	585	423	50	68

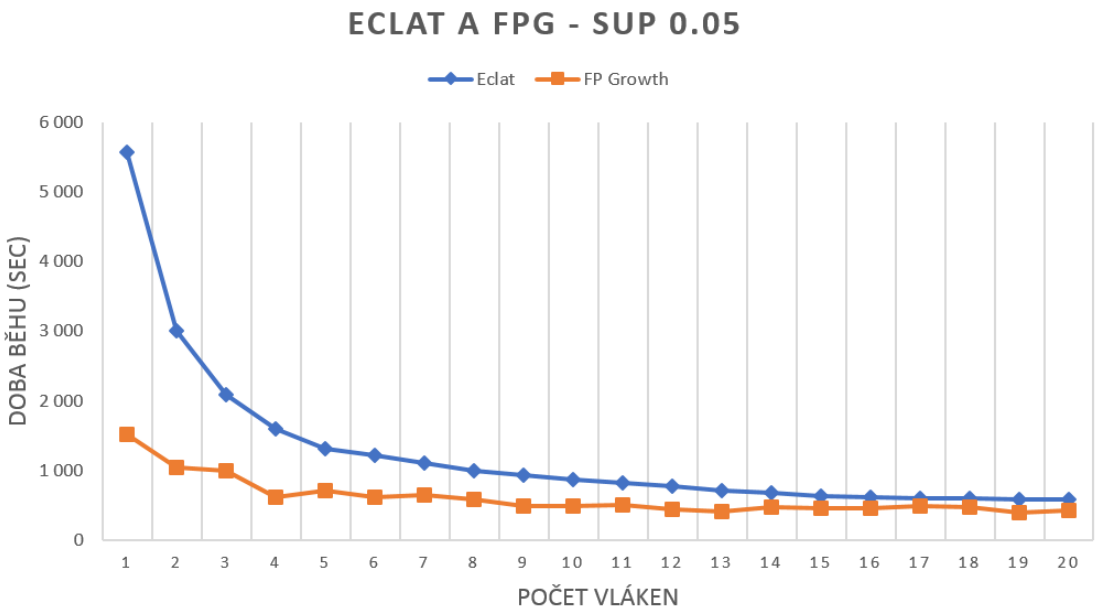
Parametry experimentu:

- **Server:** soukromý
- **Dataset:** dataset 1
- **Minimální support:** postupně 0.05 a 0.1
- **Minimální confidence:** 0.9
- **Počet vláken:** 1-20

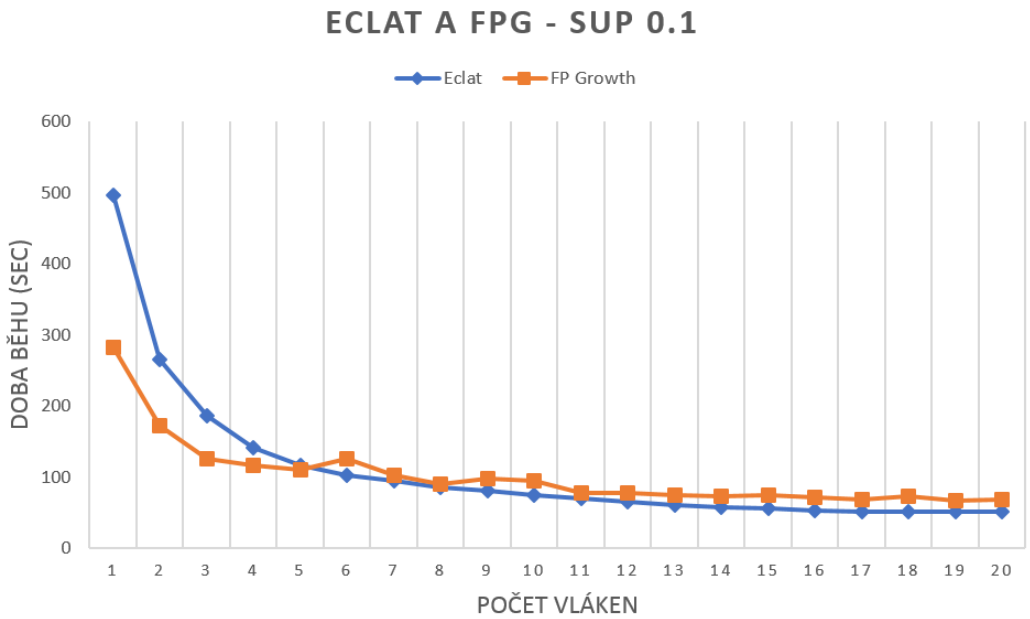
Naměřené hodnoty pro algoritmus Eclat a FP Growth jsou v tabulce 6. Hodnoty v této tabulce jsou průměrem 30 měření.

6.4.1 Vyhodnocení experimentu 2

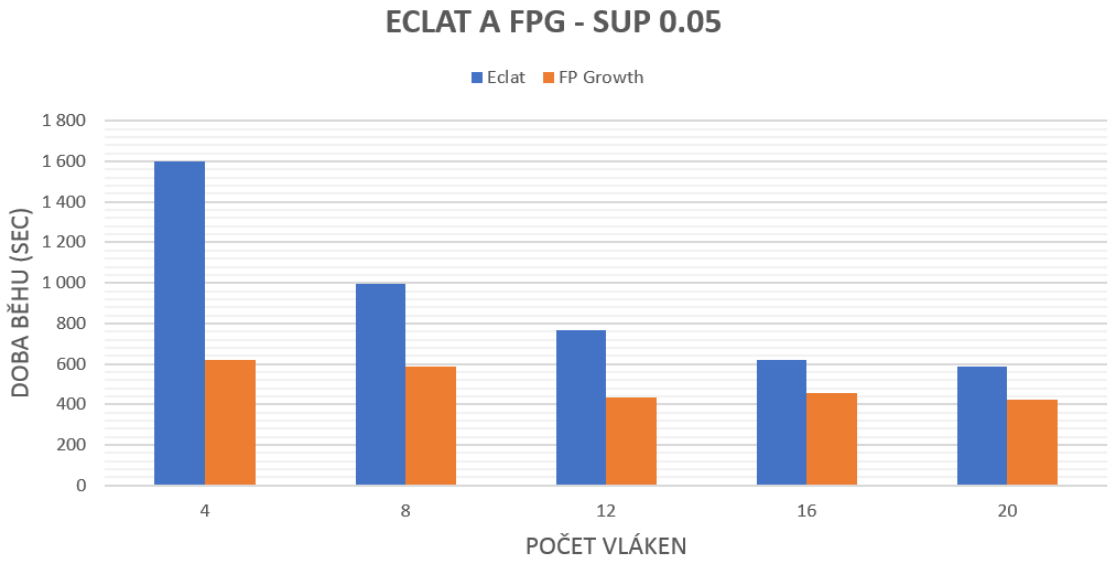
Grafické znárornění porovnání běhu obou algoritmů je na obrázcích 30, 31, 32. Z výsledků tohoto experimentu je poznat, že i vyšší počet vláken vede k dalšímu zrychlování výpočtu. Tento experiment dále potvrzuje pozorování z předchozího experimentu - FP Growth má pro nižší hodnoty supportu vyšší výkon, než Eclat.



Obrázek 30: Experiment 2 - Porovnání paralelizace Eclat vs FP Growth - support 0.05



Obrázek 31: Experiment 2 - Porovnání paralelizace Eclat vs FP Growth - support 0.1



Obrázek 32: Experiment 2 - Porovnání paralelizace Eclat vs FP Growth - support 0.05

6.5 Experiment 3

Cílem tohoto experimentu bylo vyzkoušet chování paralelních úprav na větším datasetu.

Parametry experimentu:

- **Server:** školní
- **Dataset:** dataset 2
- **Minimální support:** postupně 0.05 a 0.1
- **Minimální confidence:** 0.9
- **Počet vláken:** 1-8

Naměřené hodnoty pro algoritmus Eclat a FP Growth jsou v tabulce 7. Hodnoty v této tabulce jsou průměrem 10 měření.

6.5.1 Vyhodnocení experimentu 3

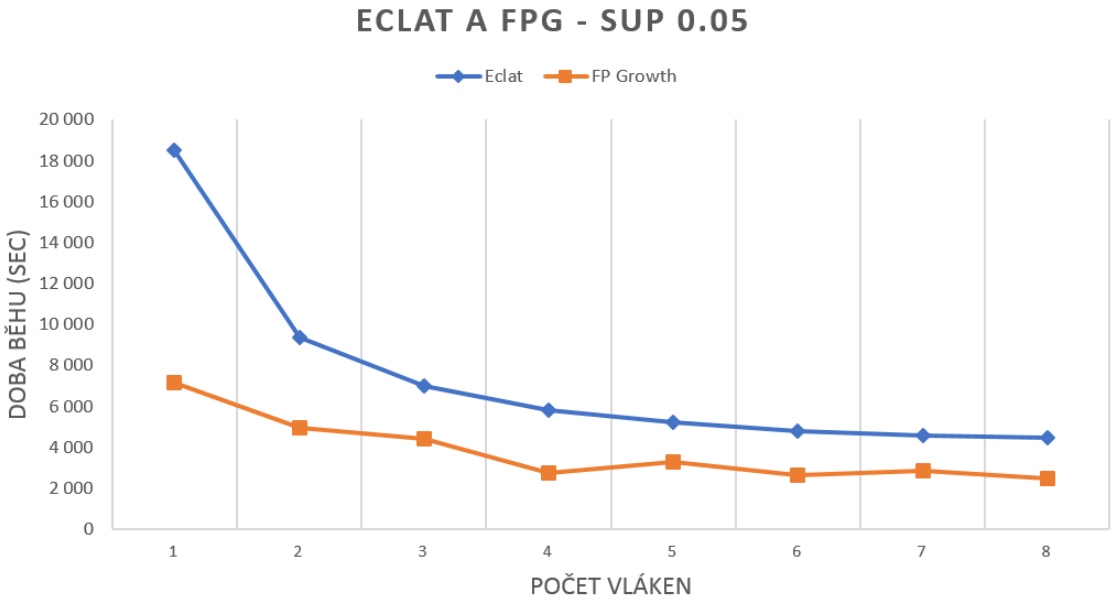
Grafické znázornění porovnání doby běhu obou algoritmů je na obr. 33. Výsledky jsou velmi podobné experimentu 2, implementace se tedy chová stejně i na větším datasetu.

6.6 Experiment 4

Cílem tohoto experimentu bylo vyzkoušet chování paralelních úprav na největším datasetu.

Tabulka 7: Experiment 3

Doba běhu (sec)	**Minimální support**			
	0.05		**0.1**	
Počet vláken	**Eclat**	**FPG**	**Eclat**	**FPG**
1	18506	7146	1567	1387
2	9376	4944	1006	797
3	7022	4407	640	546
4	5807	2729	556	513
5	5229	3290	481	493
6	4785	2651	441	539
7	4589	2835	427	510
8	4482	2498	418	385



Obrázek 33: Experiment 3 - Porovnání paralelizace Eclat vs FP Growth - support 0.05

Tabulka 8: Experiment 4

Doba běhu (min)	**Minimální support**	
	0.1	
Počet vláken	**Eclat**	**FPG**
1	284	211
2	141	132
3	123	113
4	103	91
5	99	86
6	95	96
7	94	79
8	92	73

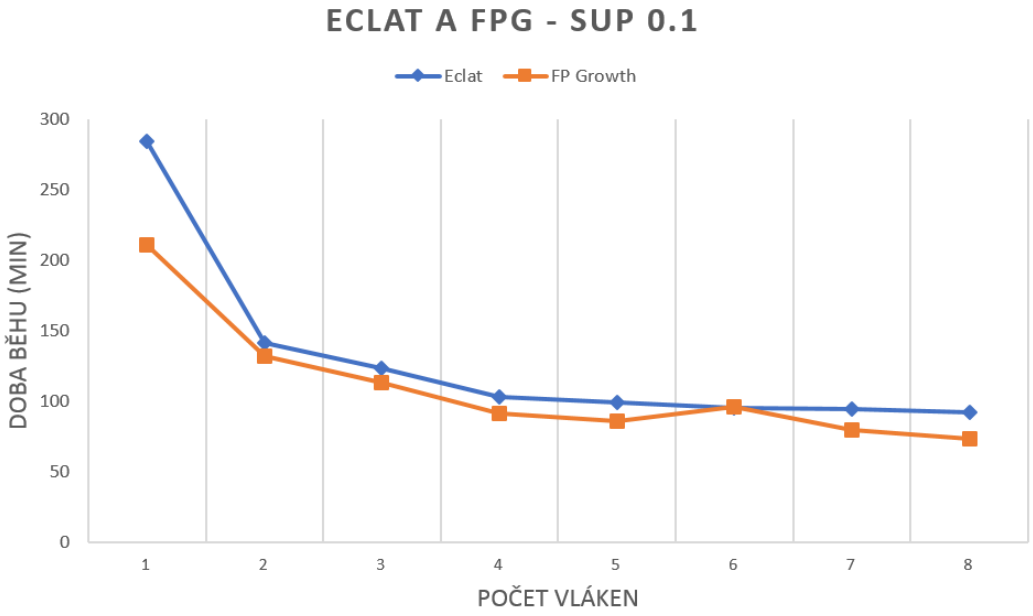
Parametry experimentu:

- **Server:** školní
- **Dataset:** dataset 3
- **Minimální support:** 0.1
- **Minimální confidence:** 0.9
- **Počet vláken:** 1-8

Naměřené hodnoty pro algoritmus Eclat a FP Growth jsou v tabulce 8. Hodnoty v této tabulce jsou průměrem 3 měření.

6.6.1 Vyhodnocení experimentu 4

Grafické znázornění je na obr. 34. Výsledky tohoto experimentu znázorňují, že i na velmi velkém datasetu se algoritmy chovají podobně jako na malém. Opakují se zde poznatky z předchozích experimentů. Ověřili jsme zde schopnost algoritmů zpracovat i velká data.



Obrázek 34: Experiment 4 - Porovnání paralelizace Eclat vs FP Growth - support 0.1

7 Závěr

Cílem této práce bylo implementovat jednu nebo více metod pro generování asociačních pravidel a upravit je pro efektivní běh v paralelním prostředí nad velkými daty. Vytvořená implementace zahrnuje dva populární algoritmy: Eclat a FP Growth - upravené pro běh v paralelním prostředí. Tyto algoritmy byly otestovány nad testovacími datasety a efektivita paralelních úprav byla těmito experimenty potvrzena. U algoritmu FP Growth se ukázalo, že je možné jej do budoucna ještě lépe optimalizovat pro paralelní prostředí. Tato optimalizace může být provedena například pomocí rovnoměrnějšího rozdělení práce mezi vlákna pomocí pokročilejších výpočtů.

Literatura

[1] Mohammed J. Zaki, Wagner Meira, Jr., Data Mining and Analysis: Fundamental Concepts and Algorithms, Cambridge University Press, květen 2014. ISBN: 9780521766333

[2] Aggarwal C.C. (2015), Data Mining: The Textbook, Springer

[3] Sujith Mohan Velliyattikuzhi, Parallel implementation of Apriori algorithm and association of mining rules using MPI, podzim 2012

[4] Mohammed J. Zaki, Srinivasan Parthasarathy, and Wei Li. A localized algorithm for parallel association mining.

[5] Osmar R. Zaiane Mohammad El-Hajj Paul Lu, Fast parallel association rule mining without candidacy generation

[6] Brijendra Singh, Rohit Miri, An Efficient Parallel Association Rule Mining Algorithm based on Map Reduce Framework, červen 2016

[7] Eibe Frank, Mark A. Hall, and Ian H. Witten (2016). The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.

A Příloha 1 - DVD

DVD s přílohami obsahuje:

- Zkompilovaná aplikace + konfigurační soubor
- Malý ukázkový dataset
- Zdrojové kódy
- Kompletní solution pro Microsoft Visual Studio 2015